Fault Tolerant Electronic System Design

POLITECNICO DI TORINO
SCUOLA DI DOTTORATO
Dottorato in INGEGNERIA INFORMATICA E DEI SISTEMI – XXVIII ciclo

Tesi di Dottorato

Fault Tolerant Electronic System Design

Boyang Du

Tutore
Prof. Luca Sterpone

Coordinatore del corso di dottorato
Prof. Matteo Sonza Reorda

Maggio 2016

Summary

Due to technology scaling, which means reduced transistor size, higher density, lower voltage and more aggressive clock frequency, VLSI devices may become more sensitive against soft errors. Especially for those devices used in safety- and mission-critical applications, dependability and reliability are becoming increasingly important constraints during the development of system on/around them. Other phenomena (e.g., aging and wear-out effects) also have negative impacts on reliability of modern circuits. Recent researches show that even at sea level, radiation particles can still induce soft errors in electronic systems.

Online error detection and Board-level functional test in processor-based system

On one hand, processor-based system are commonly used in a wide variety of applications, including safety-critical and high availability missions, e.g., in the automotive, biomedical and aerospace domains. In these fields, an error may produce catastrophic consequences. Thus, dependability is a primary target that must be achieved taking into account tight constraints in terms of cost, performance, power and time to market. Several solutions exist, acting either on hardware or software: however, they all have to face the high efforts required for designing, manufacturing, testing and qualifying processor-based systems. While standards and regulations (e.g., ISO-26262, DO-254, IEC-61508) clearly specify the targets to be achieved and the methods to prove their achievement. In this scenario, techniques working at system level (i.e., without changing the technology and the processor) are particularly attracting, especially if they can effectively meet dependability needs more efficiently without changes in the existing hardware and software.

Approaches to detect soft errors in processor-based systems are traditionally divided in techniques that deal with faults affecting the data and faults affecting the execution flow of the software application. For the faults affecting the data, to detect and eventually correct such errors in the data, i.e. *Data Error*, either detection and correction strategies can be applied to the data memory itself, such as the *Error Correction Coding* (ECC), or the software or hardware (or both) needs to be modified so that certain redundancy could be applied, for example, variable duplication plus the instructions for checking data in the software. While for the faults affecting the execution flow, although part of them are overlapped with the faults affecting the data, for example, the faults corrupting variable used in branch instruction, the rest them are difficult to handle, such as the faults affecting

the registers used in the pipeline of the processor. For mitigating soft errors affecting the execution flow, i.e. *Control Flow Error* (CFE), traditional *Triple Modular Redundancy* (TMR) could be an effective solution when it is applied at gate level of the processor, in case the netlist of the target processor is available, which is usually not the case when *Commercial Off The Shelf* (COTS) component is used, let alone the cost it introduces for verification of compliance to the standards and regulations as mentioned above, since the processor's hardware is modified. Avoiding the huge hardware overhead caused by TMR if applied at system level (>200%), solutions have been proposed to firstly detect the CFE either with extra instructions inserted into software or an extra component monitoring the processor (e.g. a watchdog processor); and then correction of CFE could be done either simply reset the processor or replying on further software techniques such as checkpoint rollback depending on the nature and requirement of workload for the processor. This work mainly focuses on online test for detecting CFEs in the first part, a hybrid solution is discussed afterwards. Since there already exists debug interface in many processors (standalone or cores), assisting designer for debugging hardware/software at different stages, which can provide information of the software running on the processor in a non-intrusive way (e.g., the debug interface in LEON3 processor), an external hardware module, namely CFC module, was proposed to be attached to the processor through debug/trace interface, to extract the information and monitor the execution of the software on the processor.

With the debug interface in the processor as LEON3, the CFC module is able to extract executed instructions and the corresponding Program Counter (PC) value. Meanwhile the software running on the processor can be divided into *Basic Blocks* (BBs), in which all the instruction will be executed sequentially without branch or jump instructions. The main idea behind the CFC module is to calculate the signature of each BB executed by the processor and compare it with the signature previously stored in the table, namely CFC Signature Table in side the CFC module.

The CFC module is greatly smaller than the processor itself in terms of area consumption. With data from simulation-based fault injection campaign on both LEON3 and miniMIPS processor with several benchmark applications, the proposed CFC module proved to be a non-intrusive, effective way for detecting CFEs without modifying the software and processor implementation.

As the CFC module focuses only on the CFEs, a hybrid technique was proposed with dual control flow monitoring to detect soft errors, together with a software-based technique targeting on Data Errors.

The hybrid technique consists of an external *Hardware Monitor* (HM) that also attaches to the debug/trace interface of the processor for extracting the same information as in CFC module. However, the HM also monitors the communication between the processor and the memory on the system bus. By extracting the processor's reading address sent to memory component, and the data it retrieves, the HM is able to get the input stream of the instructions fetched by the processor; and the information from the debug/trace interface provides the output stream of the instructions executed by the processor. Inside the HM, the input and output instruction streams are carefully synchronized and compared to detect occurrence of CFE, and a part of Data Error is also covered in this way, and in order to achieve full coverage including the Data Errors, a software-based technique,

combining "Dataflow duplication" and "inverted branches" is applied.

The fault injection campaign, emulating effects of *Single Event Upset* and *Single Event Transient*, was carried out, and the results verified the high fault coverage the hybrid technique can achieve, with a small hardware overhead.

To further exploit the existing debug interface in the processor for testing purpose, *Printed Circuit Board Assembly* (PCBA) *Power-On Self-Test* (POST) was investigated for finding feasible solution to increase processor's observability. POST plays an important role in many systems, since it may detect faults arising during the life time of the product, thus increasing its dependability. POST may use different solutions, which should match the constraints of the environment the system is deployed in.

Functional test represents a commonly adopted solution for POST. More in general, functional test is adopted in many scenarios, at the device, board and system levels. In some of them it complements other test steps, performed resorting to *Design For Testability* (DFT). The importance of this kind of defects is significantly increasing in the last years, especially since they are considered one of the major contributors for *Non Failure Found* (NFF), thus raising the interest for any solution able to improve the achievable defect coverage.

A Monitoring IP was proposed which can be mapped to a FPGA device on board, which is able to be configured to connect to the debug interface of the target processor on board (test for other devices on board is out scope of this work, and can resort other specific methods). For concept evaluation, the CoreSight Trace Infrastructure available in processors from ARM was used, which is able to provide various information related to execution of the software, such as target address of a branch instruction and information of an exception. A Monitoring IP was developed on Zynq device from Xilinx, which equips a single chip integrated an ARM Coretex-A9 dual core processor and FPGA device. By a demo project implement on Zynq device with the Monitoring IP validated that it is feasible for such an external module resides in the FPGA, monitoring the processor while a test program is being executed, to increase the observability of the processor for POST.

Since lacking of detailed information about the structure of the ARM processor clearly prevents us from computing the increase in defect coverage that can be achieved using our solution, fault simulation experiments were executed on a MIPS-like processor for which the model is available. The same information produced by the ARM debug interface is extracted from this processor during the execution of the test, and the achieved fault coverage is computed. Results show the effectiveness of the proposed solution. Interestingly, they demonstrate that the stuck-at fault coverage that can be achieved is comparable with the one reachable using a corresponding test program in a scenario where all the processor outputs can be continuously monitored.

Single Event Effects analysis and mitigation on Field Programmable Gate Array

On the other hand, *Field Programmable Gate Array* (FPGA) devices are becoming more and more attractive, also in safety- and mission-critical applications due to the high performance, low power consumption and the flexibility for reconfiguration they provide. Two types of FPGAs are commonly used, based on their configuration memory cell technology, i.e., SRAM-based and Flash-based FPGA. Besides the hardware resources for I/O, clock managing and on-chip memory etc., FPGA device can be modeled as an matrix of Logic Block and Switch Box connected by interconnection segments of various lengths. The Logic Block contains configuration logic resources such as *Look Up Table* (LUT), Multiplexes and Flip-Flops for implementing different user logic functions. While the Switch Box (or Switch Matrix) contains programmable interconnection segments user can configure as active or inactive to form different interconnection networks among Logic Blocks and other hardware resources for routing of the implemented design. And the configuration memory of FPGA device holds the configuration data of all the resources to be used in the design, including the Logic Blocks and Switch Boxes. When the FPGA device is used in harsh environment, such as in space and avionic applications, fault tolerant strategies must be applied as highly charged particles could induce *Single Event Effects* (SEEs) in configuration memory and/or user logic leading to system misbehavior.

For SRAM-based FPGA, since SRAM cells are highly susceptible to radiation induced effects and one bit corruption inside the configuration memory may leads to drastic change in the logic mapped on the FPGA, certain fault tolerant strategy needs to be applied when SRAM-based FPGA is used especially in safety- and mission-critical applications. Traditional techniques include Triple Modular Redundancy (TMR) and configuration memory scrubbing.

The TMR technique can be carried out at different levels, for example, at gate level, logic path level, entity level and even system level. However as the name suggested, TMR introduces large resource overhead, including area and power consumption. So to trade off the reliability and overhead, different variations were proposed, for example, TMR on selective gates, TMR with approximate logic etc.

In this part of the work, the *Verification and Error Rate Integrated* (VERI-Place) tool was used for error rate prediction and mitigation on SRAM-based FPGA. The VERI-Place tool takes files from standard commercial flow (from Xilinx) as inputs, and generates report including information regarding the sensitive part of the target design and error rate when *Single Event Upset* (SEU) accumulated in the configuration memory. Furthermore, the tool acts on the Place & Route step of the design, without introduce extra hardware overhead into the design, to improve the reliability of the circuit to be mapped on the FPGA.

The accuracy of the error rate prediction and effectiveness of the SEU mitigation made by the VERI-Place has been verified in simulation analysis and two radiation experiments with two benchmark circuits, including an ARM-based SoC, in two different facilities with different radiation profiles. The design version generated by the VERI-Place tool based on the version with *Xilinx TMR* (XTMR) applied is able to achieve an improvement of

reliability by two orders in terms of *Silent Data Corrupt* (SDC) cross-section.

For Flash-based FPGA, even though their configuration memory made of non-volatile flash cells that are almost immune to SEU, the floating-gate-based switches and the Flip-flops in the configurable logic can still suffer from SEEs induced by radiation particles.

Regarding Flash-based FPGAs, two distinct effects may be identified. The former occurs inside of the floating gate switch: the pass transistor and floating gate transistors usually constitute the floating gate switch. The second occurs when a high charged particle hits a sensitive node of a logic cell belonging to the FPGA's configuration tile. The generated pulse may propagate through the logic depending on the FPGA tile configuration. If the tile is configured to implement a latch, the pulse may turn directly into a SEU because of the feedback paths implemented by the tile logic configuration. Meanwhile if the tile is configured to implement a logic gate, the transient pulse is assumed to be propagated only if the voltage glitch generated by the particle hit on the struck node changes by more than $V_{DD}/2$. Once a SET is generated into the sensitive area of a logic gate it starts its propagation through the logic paths until a sequential element is reached. During its propagation the SET pulse may pass through inverting and non-inverting gates. The SET propagation through logic gates undergo to different electrical phenomena that affect the shape of the pulse modifying its voltage amplitude, the width and the speed along the traversed logic path.

An analytical SET model was developed for accurately investigate the SET behavior when it propagates through the logic paths in the design. Depending on the type of gate and its input and output load, a SET pulse could be filtered or broadened when it traverses through the gate. The model has been verified with the data collected in the SET analysis experiment via electrical pulse injection carried out before.

Furthermore, a SET-Analyzer (SETA) tool was used for analyzing the SEE sensitivity of several benchmark circuits including a RISC microprocessor from OpenCores. The SETA tool takes netlist and placement files generated from the commercial tool (Libero SoC in case of Microsemi FPGA), generates a set of transient pulses, and with the analytical SET model, it propagates the pulses along the logic paths in the design. SEE sensitivity information regarding the probability of SET pulses reaching a Flip-flop and eventually sampled is reported by the SETA tool. With this information, designer can determine which FFs are critical and sensitive, and need to be protected using techniques such as *Guard Gate* (GG) insertion as proposed. The GG is a configurable circuit macro that can be inserted at the input of a FF for filtering a user-defined length SET pulse. Depending on the output of SETA tool, designer can selectively choose to insert GG structures to the design.

Finally, a SET-aware place and route tool (SET-PAR) was used to generate an improved version of the target design regarding SET sensitivity, by acting on the placement and routing resources optimization for SET filtering. The SET-PAR tries to place the gates with SET filtering effect close to each other and the gates with broadening effects far from each other to reduce the probability of a SET reaching and sampled by the FF, i.e. reducing the SET sensitivity of the target design, while respecting the timing constraints.

With all the models and tools mentioned above, a complete SEE analysis and mitigation flow for Flash-based FPGA was proposed. Since the flow takes the files from

commercial tool as input and generates improved design as constraint files, it can be easily integrated into standard commercial toolchain, and with analysis of simulation experiments over several ITC99 benchmark circuits and a heavy-ion radiation experiment carried out on Microsemi ProASIC3 Flash-based FPGA with a RISC microprocessor, the propose flow has been proven to be accurate in the SEE sensitivity analysis and effective in SEE mitigation without introducing extra hardware overhead and performance degradation.

Acknowledgements

I would like to thank Prof. Luca Sterpone and Prof. Matteo Sonza Reorda for providing me the opportunity to pursue my research activity in the CAD group, and for their great support and generous help over the three pleasant years of PhD.

I also would like to thank all the people I had the chance to have collaboration with for the thing I have learned from them and the results we achieved together along the way, and also all my colleagues in the CAD group and all the people in the Lab3 for the pleasant time we spend together.

Finally, I would like to thank all my friends for the companion they provide me over the years starting from the very beginning when I came here in Italy, and all my families for the support and love they provide me generously and constantly.

Contents

List of Figures

List of Tables

Part I

Testing for SoC/SoPC by Exploiting Debugging Infrastructures

Chapter 1

Introduction

Facing the effects of faults in electronic systems is an increasingly important issue, especially when they are used in safety- or mission-critical applications. Reduced transistor size, higher density, lower voltage and more aggressive clock frequency may increase the susceptibility to soft errors to unacceptable levels. Other phenomena (e.g., aging and wear-out effects) also impact negatively the reliability of modern circuits. As a consequence, the need for effective techniques providing the ability to detect the possible occurrence of faults during the operational phase is becoming a major issue from a practical point of view.

Processor-based systems are commonly used in a wide variety of applications, including safety-critical and high availability missions, e.g., in the automotive, biomedical, telecommunication and aerospace domains. In these fields, an error may produce catastrophic consequences. Thus, dependability is a primary target that must be achieved taking into account tight constraints in terms of cost, performance, power and time to market. Several solutions exist, acting either on hardware or software: however, they all have to face the high efforts required for designing, manufacturing, testing and qualifying processor-based systems. Standards and regulations (e.g., ISO 26262 [34], DO-254 [10], IEC 61508 [33]) clearly specify the targets to be achieved and the methods to prove their achievement. In this scenario, techniques working at system level (i.e., without changing the technology and the processor) are particularly attracting, especially if they can effectively meet dependability needs more efficiently without changes in the existing hardware and software. Solutions based on additional modules that monitor the processor behavior and check its evolution looking for possible fault effects belong to this category.

1.1 Online test of Control Flow Error

Among the techniques that can be applied at system level, *Control Flow Checking* (CFC) is particularly effective. Control Flow Checking consists in verifying the sequence of instructions executed by a processor. It must be noted that CFC alone cannot detect data errors, i.e., errors in data registers or data memory, and must be generally complemented by some technique focused to data error detection. However, a significant percentage of

errors in a processor usually manifest themselves as *Control Flow Errors* (CFE), and most data errors can be effectively addressed by adding proper fault management mechanisms to memory and relevant registers (e.g., parity check or hamming codes).

1.1.1 Previously proposed techniques

Several CFC approaches are based on signature monitoring [60]: the program is divided into a set of blocks (named basic blocks), having only one entry-point and only one exit-point: hence, whenever the entry-point instruction is executed, the following instructions in the block are executed. Each basic block has an associated signature that is calculated at compile time and stored in the system. During the operational phase, a run-time signature is calculated and (at the end of the block execution) compared with the reference signature, thus allowing to detect any error affecting the block execution flow.

Signature monitoring techniques usually require software and/or hardware modifications to calculate the run-time signatures and perform comparisons. Since the calculation of signatures is computationally expensive, dedicated hardware (e.g., a *watchdog processor*) is sometimes used for this purpose. In some cases the approach is combined with software modifications, which support the watchdog processor operation. The *Disjoint Signature Monitoring* (DSM) approach [35, 15] solves these problems by storing reference signatures in an auxiliary memory and using a watchdog processor to compare the reference signatures with the run time signatures. This approach does not require support from the software program. However, a drawback of the DSM approach is that the watchdog processor does not have access to the internal operation of the processor, which leads to a lower error coverage, particularly with complex processors [15].

1.1.2 Control Flow Checking module

A new method was proposed for the control flow checking approach resorting to available debug infrastructures as a means to monitor the processor behavior. Debug infrastructures are intended to support software debugging in embedded system development, and are very common in modern processors. Since they are useless when the operational phase is entered, they can be easily reused for on-line monitoring in an inexpensive way [8, 29, 54, 56]. On the other hand, they can provide internal access to the processor without disturbing it and do not require any modification either to the processor or to the software running on it. It must be noted that in some processors the debug information is processed and compacted by a specialized module, such as LEON3's Debug Support Unit (DSU) or ARM's Embedded Trace Module (ETM). Such modules are not used in our approach, which is based on direct access to the trace interface.

In the proposed approach, control flow checking is performed by an external hardware module that monitors the sequence of instructions executed by the processor through the debug interface and performs some checks to detect possible deviations. The developed hardware module, named *Control Flow Checking Module*, consists of a core able to perform some checks at the end of each basic block, using the information provided by the debug interface. Along the work, different versions of CFC Module were proposed providing possibility for designer to choose the trade-off between the detection capability and the

cost in terms of area consumption. The detection capability of the developed module has been experimentally evaluated by means of fault injection on two different pipelined processors.

1.2 A hybrid nonintrusive error detection technique

The CFC module proposed can effectively detect Control Flow Errors, however to detect Data Errors, such solution must be accompanied by other techniques. So a new method was proposed to use dual control-flow monitoring for nonintrusive error detection, combing software techniques with an external hardware module that monitors the execution of a microprocessor.

The external hardware module used in this method is similar to the CFC module in the way that it also monitors the downstream information extracted from the trace interface of the processor (e.g., LEON3 [27]), however, it also captures the upstream communication at the bus between the memory and the microprocessor. If an error corrupts the instruction flow at any stage, it is detected by comparing the downstream instruction flow with upstream instruction flow. On the other hand, errors in the generation of fetch addresses are detected using a PC prediction technique. In this way, all errors in the program counter and the instruction register at any of the pipeline stages can be detected. These include all control-flow errors, as described in [76]. It can also detect some data errors produced by corrupted instructions that generate wrong data or data addresses without affecting the control flow.

To achieve full error coverage, it is necessary to protect data as well as control flow. A combination of two software-based techniques

- total data-flow duplication, based on techniques presented in [17] and [59]
- inverted branches, based on the approach presented in [8]

were chosen for data hardening to lessen the impact of data-flow hardening in the hardware module since hardware-based data-flow hardening requires additional connections to the microprocessor architecture that are not easily available, and the hardware implementation leads to area increase.

1.3 Print Circuit Board Assemblies Power-On Self-Test

Besides being used for online test of CFE, the debug infrastructure can also be exploited during the Power-On Self-Test (POST). POST plays an important role in many systems, since it may detect faults arising during the life time of the product, thus increasing its dependability. POST may use different solutions, which should match the constraints of the environment the system is deployed in.

1.3.1 Functional test for POST

Functional test [67, 65] represents a commonly adopted solution for POST. More in general, functional test is adopted in many scenarios, at the device, board and system levels

[39]. In some of them it complements other test steps, performed resorting to Design For Testability (DFT). For example, when considering Printed Circuit Board Assemblies (PCBAs) test, it is common to see functional test as the last step, mainly targeting dynamic defects. The importance of this kind of defects is significantly increasing in the last years, especially since they are considered one of the major contributors for NFF [37, 41], thus raising the interest for any solution able to improve the achievable defect coverage.

Hence, functional test plays a key role for in-field test of PCBAs [26]. In this context, functional test is particularly attractive due to its relatively low implementation cost (no equipment required), low intrusiveness (no or limited changes in the PCBA design and low impact on the application) and flexibility (since functional test is typically based on forcing the processor to execute a suitable test program, possibly mimicking some specific application code). A commonly adopted solution consists in launching at the power-on the execution of a functional test targeting all the critical modules in the system, or, alternatively, some specifically crafted test able to make observable the highest percentage of defects [55]. Functional test is normally based on a sequence of instructions to be executed by the in-system processor(s), aimed at exciting possible defects in the processor itself or in any other device on the board, as well as in the interconnect. Therefore, this kind of test is sometimes referred to as *Software-Based Self-Test* (SBST) including [57].

Major limitations to the effectiveness of functional test include

- The difficulty in assessing the achieved defect coverage: although many efforts have been done to introduce high-level metrics, their correlation with real defect coverage is still a matter of discussion and a hot research topic [36];
- The cost for creating suitable functional stimuli; a lot of work has been done to automate this part, or at least to provide guidelines for the test engineer in charge of developing suitable functional tests;
- The limited observability that can be obtained on the behavior of the system under test, both as a whole and in terms of its components.

1.3.2 Monitoring IP

As in the CFC Module, a Monitoring IP mapped into FPGA device on the same board of the target processor can be used to extract information via Debug Interface of the processor for assisting functional test. The onboard FPGA (if exists) can be configured as the Monitoring IP during the POST, and reconfigured to carry out normal operations afterwards. The implemented Monitoring IP is targeted on ARM processor (to be precise, Cortex A9 on Zynq-7000 device), whose CoreSight components can generate various information packets related to the execution of the software on the processor, which then can be extracted from the Trace Port Interface Unit (TPIU).

However, internal characteristics of the ARM processor is not directly available that makes impossible to execute fault injection to verify the effectiveness of proposed approach. Instead, a pseudo debug interface was added in a pipelined processor mimicking the behaviors of the one in ARM processor.

Chapter 2

Online Test of Control Flow Error

This chapter discusses about the technique proposed for online test of *Control Flow Error* (CFE), i.e. the CFC module[1]. The CFC module is used to exploit the debug/trace interface, that already exists in many processors, to monitor the software running on the processor for detecting CFEs.

The chapter is organized as follows: firstly, a background section is presented providing information regarding soft errors in processor-based systems, the existing debug interface and related approaches proposed to detect the CFEs; then the proposed technique involving the CFC module is presented explaining different solutions enabling designer to achieve trade-off between hardware overhead and test capability; finally, the data analysis from fault injection campaigns on two target processors is presented, showing that even with very low hardware overhead and no performance penalty, the proposed CFC module is able to achieve very high test coverage with respect to CFEs.

2.1 Background

Approaches to detect soft errors are traditionally divided in techniques that deal with faults affecting the data and faults affecting the control flow. In order to mitigate errors affecting data, the common approach is duplication. Duplication can be accomplished at different levels: computation duplication, procedure duplication and program duplication. Many of the methods to detect soft errors affecting data require some software modification. Because of the required duplication and additional data checks, a considerable performance decrease is produced, as illustrated in [17] and [23].

Several approaches have been proposed to detect and mitigate soft errors affecting the control flow. The most common approach is known as Embedded Signature Monitoring (ESM) [60]. Variations can be applied to this technique but the basic idea consists in pre-computing a signature for certain parts of the code and then re-computing the same

[1]This work was done with collaboration with our colleagues in University Carlos III of Madrid, Leganes, Madrid and supported in part by EU FP7 STREP project BASTION; related publications can be found as [20, 21, 22]

signature at run time, checking whether the two match. In these techniques, code is divided into sections called *Basic Blocks* (BBs). A BB is a fragment of code that does not contain any incoming or outgoing branch, i.e., all the instructions belonging to the BB are always executed sequentially, once the first is reached.

In *Enhanced Control flow Checking using Assertions* (ECCA) [3] ESM is implemented through assertions. In each block, two assertions are included: one at the beginning of the block which assigns an identifier to the block and checks the correctness of the branch previously taken and the other at the end of the block, which updates the identifier and checks it. In this technique, short blocks increase the error detection capability as well as decrease the latency. The main drawback of this technique is the high memory and performance overhead it introduces.

In *Control Flow checking by Software Signature* (CFSS) [51], a signature is assigned to each block. At runtime, a signature is continuously computed out of the opcodes of the executed instructions: each time a branch takes place the signature is compared with the expected one. Signatures are computed in different ways depending on the number of predecessors the block has. The main limitation of this method lies in its limited detection capability, since some errors (e.g., when the processor wrongly executes another instruction of the same block, or when a legal but incorrect branch is taken) are not detected.

In *Control flow Error Detection through Assertions* (CEDA) [75], the signatures are computed in different ways depending on the block type. For each block two signatures are assigned (the signature at the beginning and the signature at the end of the block). Check instructions are inserted in certain points of the code to verify the correctness of the signature.

In *Automatic Correction of Control-flow Errors* (ACCE) [76], a software method allows not only to detect, but also to correct control flow errors.

Summarizing, all the described techniques compute one or several signatures and compare them with the ones previously assigned and stored. Mismatches between signatures trigger an error signal, and the possible correction. Detection problems can arise when incorrect branches are taken inside a block or when legal but incorrect branches are taken. As a major limitation, most of the methods involve a significant overhead in terms of memory and performance.

Other methods, essentially based on an additional monitor processor, like the approaches presented in [35, 15, 13] require a large overhead of hardware resources to detect the soft errors; moreover, the detection capability is not independent from the application. Thus, not surprisingly, in these methods the error detection capability increases with the complexity of the additional hardware.

Another drawback of these methods is the difficulty in observing the processor behavior from the outside. In [15] a watchdog processor connected to the cache memory is used to detect soft errors. This watchdog processor uses a combination of techniques that are applied when a block ends or when frequent instructions are executed. For intra-block instructions a special signature is computed and address checking is performed. For non-conditional branch instructions the destination address is compared with the information about the code stored previously, while for conditional branch instructions, the watchdog processor verifies the correctness of the taken branch. Exceptions are also taken into

account by accessing the interrupt vector. Data errors are handled by duplicating only critical variables. Although the error coverage is incremented, the proposed watchdog processor is quite complex and requires a large amount of resources for its implementation.

Some major limitations of these methods are the following:

1. The presence of caches prevents the watchdog to know what the processor is exactly doing, and may significantly decrease the detection capabilities of the approach.

2. If the processor is based on a pipeline architecture, observing the bus during the fetch operation does not allow to understand which instructions are really executed, due to the effects of branches.

In [8] the authors proposed a method to implement some checks (partly in hardware, partly in software) able to detect both data and control flow errors; according to the authors the method is able to detect all possible faults, but it requires some relevant modifications on the application software, and it assumes that the processor bus is accessible, which is not the case when caches exist.

The usage of debug infrastructures has been proposed as a way to increase the observability of soft errors with very limited latency, as presented in [29, 54, 56]. The technique proposed in this work follows the same approach, taking advantage of the already available features existing in many processors (standalone or cores) and using them to detect soft errors. In particular, the method exploits the feature offered by some debug architectures, which allows tracing the values of both the Program Counter and the Instruction Register during the execution of an application. In order to exploit on-the-fly these information, an external hardware module is added, which monitors the values provided by the debug port and processes them for detection purposes. In this way the processor is not modified and the connection to it can be made in an easy and efficient way, while caches do not represent any more an obstacle.

In [29] the debug infrastructure is exploited to get the value of the Program Counter of two processors running the same application, checking for possible divergences caused by faults. The technique explored in [56] further extends this idea, and assumes that either time, or hardware redundancy is used, and the debug interface allows checking for discrepancies between the two execution replicas. An extension of this idea is presented in [54] where the authors evaluate the effectiveness and cost of different checks that can be performed on the values stemming from the debug port.

2.2 Control Flow Checking module

The CFC module was proposed as a technique for online test of CFE in which:

- the address and machine code of each executed instruction are available through the debug port

- these two information items processed on-the-fly by a suitable hardware module (i.e., *CFC module*) aiming at detecting possible CFEs affecting the processor.

The CFC module can be implemented either as an Infrastructure IP, if *System-on-Chip* (SoC) is considered, or as an additional device (if board systems are considered): in the latter case it could also be implemented on an FPGA located close to the processor. The CFC module implements some checks aimed at detecting control-flow errors and triggers an Error signal when it detects any error. The architecture of the resulting system is shown in Fig. 2.1.

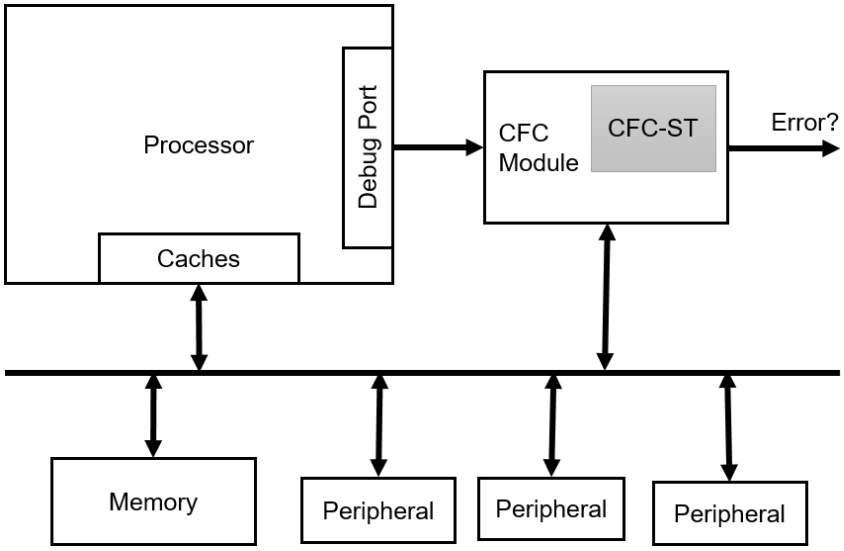


Figure 2.1: Architecture of a system adopting the proposed approach

The proposed method for CFE detection requires that the target processor be equipped with a debug port interface available for the external CFC module to link with; moreover, the method assumes the debug port provides a continuous flow of information about instructions executed (i.e., completed) by the processor, skipping instructions which are fetched, and then aborted (due for example to branch effects). This assumption is justified by the fact that many of current existing processors have a trace bus (such as the ones compliant with IEEE ISTO 5001-1999 Class 2+).

2.2.1 Architecture of the CFC module

In summary, there are two checks performed by the CFC module:

1. *Check #1*: at each clock cycle, the CFC module receives the address and machine code of the currently completed instruction (if any): it checks whether the address is correct with respect to the previous instruction. In particular, if the previous instruction was a branch, the CFC module decoded it and computed the target address T : hence, the address X_i of the current instruction i should be equal either to T (branch taken) or to $X_{i-1} + IS$ (branch not taken), being X_{i-1} the address of the previous instruction, and IS the instruction size in bytes.

2. *Check #2*: the CFC module continuously compacts the machine codes of the executed instructions: each time a branch instruction is executed, a BB ends, and the module compares the computed signature for the block with the expected one. The signature is computed by compressing into a single value the machine codes of the instructions belonging to the BB.

The CFC module supports the executions of both Check #1 and Check #2. The implementation of a hardware module supporting Check #1 requires some proper logic, able to remember some proper information about the previous instruction, as well as to perform the proper comparison between the new and expected values of the PC.

However, to support the Check #2, a table inside the CFC module is implemented to store the pre-computed signatures of BBs, i.e. the *CFC Signature Table* or *CFC-ST*. In the operational time, each time a BB starts, the CFC module records internally the address of its first instruction (*Starting Address* or *SA*); when the BB ends, the module retrieves the expecting signature of the BB from CFC-ST using the least significant bits of SA as an index, and compares it to the signature computed on-the-fly by the Signature Monitor component. If a mismatch is found between the two signatures, an Error signal is triggered.

Along the work, different versions of CFC module were proposed focusing on how the signatures of the BBs in the software running on the processor should be stored and managed to enable the trade-off between the detection capability of the CFC module and resource consumption overhead.

The CFC module with static CFC-ST

On the assumption that the software running on the processor or at least the critical part of the software is small, in terms of number of BBs, so that all the signatures can be stored directly in the CFC-ST, the architecture of the CFC module is quite simple as illustrated in Fig. 2.2.

The CFC module consists of the following blocks:

- *Instruction Decoder* (ID) block detects when a new instruction is being executed and checks whether that instruction is a branch instruction or not, in order to identify the beginning and end of a BB execution,
- *Signature Monitor* computes the signature of the executed machine code values by means a MISR (Multiple Input Shift Register),
- *CFC-ST* stores the off-line signatures corresponding to each BB and its SA,
- *Control Block* is in charge of managing the CFC module behavior, enabling and clearing the signature monitor for each BB, reading the off-line signature and triggering the Error signal when the on-line signature differs from the expecting one in CFC-ST. It is also in charge of implementing the Check #1.

On the other hand, in case of the CFC-ST is not large enough to hold all the signatures of BBs in the software, then certain mechanism needs to be applied to choose which

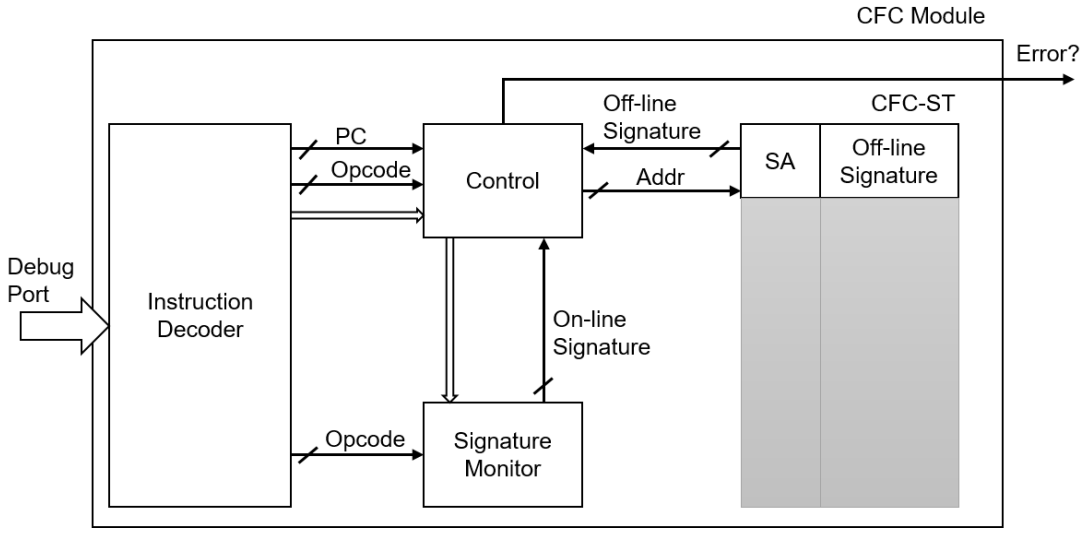


Figure 2.2: CFC module architecture diagram

signatures should be stored in the CFC-ST. Three techniques were proposed for ranking the BBs:

1. *Technique #1*: BBs are ranked according to their size (in terms of instructions). The information about the N_{CFC-ST} top ranked BBs is then stored in the CFC-ST. The rationale behind this technique is that that chance for a fault to affect a BB is proportional to its size.

2. *Technique #2*: Some execution runs are performed, applying to the application program inputs some values, which are representative of its "typical" behavior. BBs are ranked according to the number of times they are executed. The information about the top ranked BBs is then stored in the CFC-ST.

3. *Technique #3*: Some execution runs are performed, applying to the application program inputs some values, which are representative of its "typical" behavior. BBs are ranked according to the number of times they are executed, times the number of instructions composing each of them. This ranking combines the previous two. The information about the top ranked BBs is then stored in the CFC-ST.

The effectiveness of each technique is evaluated with fault injection simulation, and reported in the experimental results section.

Dynamic CFC module

To further improve the detection capability of the CFC module in the circumstance that the CFC-ST is size-limited with respect to size of the software,the Dynamic CFC (Dyn-CFC) module was proposed, in which the CFC-ST is updated on-the-fly. The CFC-ST's architecture is also modified to support the update of signatures as shown in Fig. 2.3.

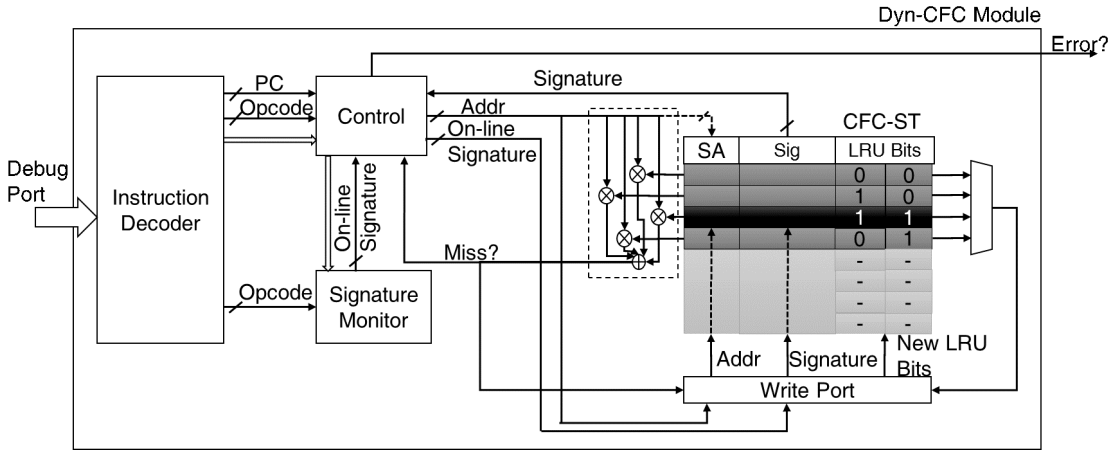


Figure 2.3: Dynamic CFC module architecture diagram

Different with the static version, the Dyn-CFC module does not require the signatures of the BBs in software to be pre-computed and stored into the CFC-ST. When the system is running, the Dyn-CFC performs the Check #2 described previously in a different way as shown in Fig. 2.4.

```
1.  dyn_cfc(){
2.      if (newBB = getBBfinished()){
3.          sig_s = findBBsigin_st(newBB.startaddr);
4.          if (sig_s == NULL)
5.              insertBBsigin_st(newBB.startaddr, newBB.signature);
6.          else {
7.              if (sig_s != newBB.signature)
8.                  raise error;
9.          }
10.     }
11. }
```

Figure 2.4: CFC-ST update mechanism

The CFC-ST is empty when the application starts, During the application execution, when the Dyn-CFC module detects the end of a BB, it searches for the corresponding BB signature in the CFC-ST:

1. if such signature is found, a comparison is carried out to possibly detect the occurrence of a CFE;

2. otherwise (this is called a *miss* event), no check is performed on the BB, and the Dyn-CFC module stores the BB signature just computed at runtime into CFC-ST for future references.

In this way, the Dyn-CFC module is able to detect a CFE affecting a BB, provided that the BB is executed at least twice before the corresponding signature is removed (overwritten) from the CFC-ST. Due to the principle of locality [32] and to the existence of cache-aware compiler optimization techniques [40], the Dyn-CFC module is able to achieve a high capability even if the CFC-ST size is significantly smaller than the number of BBs in the application.

To cope with the modified Check #2 implemented in Dyn-CFC module, taking inspiration from the architecture of cache memory, the CFC-ST is organized into sets. Each set in CFC-SET can store 2, 4 or as the designer's choice BBs' signatures. With a BB's SA as SA_i, it can only be stored into the set whose index equals to $(SA_i >> log_2(N_{CFC-ST-SET}))/N_{CFC-ST}$ where N_{CFC-ST} is the number of sets in CFC-ST and $N_{CFC-ST-SET}$ is the size of set in the CFC-ST.

When a *miss* event happens, the Control logic in Dyn-CFC will insert the signature of just finished BB into CFC-ST. Two situations could happen: 1) if the set still have empty slot, then the SA and signature of the BB is directly store into the empty slot; 2) otherwise, the Control logic has to choose which entry in the set should be overwritten. And replace policy applied in the Dyn-CFC module will affect the detect capability since it can only detect the CFE in a BB when its signature is still in the CFC-ST before it is overwritten. Again, similar to the replace policy used in cache memory, the Least Recently Used (LRU) was adopted for updating the entries in CFC-ST to take advantage of the principle of locality.

To verify the effectiveness of the CFC module, simulation based fault injection campaign was carried out on two pipeline processors, reported in following section.

2.3 Experiment results with fault injection

Two processors were used during the fault injection experiments, and fault model proposed in [76] was used as shown below. Several applications were selected as benchmark applications on both processors.

1. *Fault Model #1*: a randomly chosen branch instruction is changed into a NOP instruction.

2. *Fault Model #2*: a randomly chosen bit in the PC value is flipped at a random time.

3. *Fault Model #3*: a randomly chosen bit in the operand of a branch instruction is flipped at a random time.

2.3.1 Experiment setup

The first processor used is the miniMIPS [30] from openCores, adding a debug port which provides the address and machine code of the executed instruction. The miniMIPS' architecture is based on 32-bit registers and addresses, and includes 5-stages pipeline, accounting for about 45k equivalent gates when synthesized (with multiplier) with the FreePDK45 Generic OpenCell Library from NanGate [47].

The second processor used is the LEON3 [27] processor which implements the full SPARC V8 standard and is widely used in space applications. The LEON3 core has the following main features: 7-stage pipeline with Harvard architecture, separate instruction and data caches, hardware multiplier and divider, on-chip debug support and multiprocessor extensions. The register file is divided in a configurable number of register windows, so that at any one instant a program sees 8 global integer registers plus a 24-register window. The number of register windows is implementation-dependent and can be configured within the limit of the SPARC standard (2-32), with a default setting of 8. A basic configuration has been built to perform a fault injection campaign. The system configuration includes one LEON3 integer unit with 8 register windows, instruction and data caches (2 kB each), instruction trace interface, interrupt controller, system bus (AMBA), memory controller and general purpose input/output. Using the same synthesis design flow and library used for miniMIPS, the size of the LEON3 model is about 150K equivalent gates.

The LEON3 model includes a debug interface which perfectly fits the requirements of our method, and allows the continuous tracing of both the PC and IR registers. While for miniMIPS, in order to mimic the behavior of a debug interface, which is not available in the original miniMIPS model, its VHDL code was modified so that the address and machine code of the instruction which is currently being executed (i.e., which is in the EXE stage) are available.

Then the CFC Module was implemented in VHDL, customized it in two versions to make it suitable for usage with each of the two processors and synthesized the two versions in the same way as above. The costs for the two CFC (Dyn-CFC) modules (excluding the CFC-ST, which is implemented as a memory) amounts in about 800 and 2300 equivalent gates, corresponding to less than 2% and 1.4% of the total hardware size of the miniMIPS and LEON3 processor, respectively (the CFC module with static CFC-ST is even smaller).

The benchmarks used in the fault injection experiments include:

1. *Bubble* implements the Bubble Sort algorithm on a vector composed of 8 integer elements,
2. *Matrix* computes the multiplication between two 3 by 3 integer matrices,
3. *Dijkstra* implements the Dijkstra shortest path searching algorithm on a weighted graph with 9 nodes,
4. *RLE* implements the Run Length Encoding and Decoding algorithm on a data set composed of 100 integers (of two different values),
5. *MF* implements the Ford-Fulkerson algorithm which computes the maximum flow in a flow network (of 32 nodes connected with at least 64 random edges).

Table 2.1 reports the characteristics of the benchmarks.

2.3.2 Fault injection results

For the fault model mentioned before, 10,000 faults (we used 10,000 fault injections as it provided good confidence in the results of error classification and test coverage) were

Table 2.1: Benchmark applications

	Size[#instr]		Duration[#cc]		BBs[#]	
	miniMIPS	LEON3	miniMIPS	LEON3	miniMIPS	LEON3
Bubble	39	26	710	867	12	19
Matrix	45	61	1,240	1,500	19	20
Dijkstra	113	147	4,174	2,888	31	24
RLE	229	257	8,567	7,699	58	43
MF	1,082	744	456,560	401,332	253	172

injected for each of them and classified the effects of the injected faults as in Table 2.2, where the "Wrong Results" denotes the faults cause wrong data in the memory at the end of application comparing to the fault-free execution.

Table 2.2: Fault injection results

	Fault Model	# Injected Faults	CFEs (miniMIPS)		CEFs (LEON3)	
			Total	Wrong Results	Total	Wrong Results
Bubble	#1	10,000	6,173	4,145	7,306	3,122
	#2	10,000	7,377	4,148	8,078	5,243
	#3	10,000	6,136	2,453	3,623	623
Matrix	#1	10,000	9,252	6,079	9,747	5,232
	#2	10,000	8,305	6,908	7,267	5,042
	#3	10,000	9,283	3,421	5,855	4,577
Dijkstra	#1	10,000	6,175	3,069	9,161	3,683
	#2	10,000	7,792	3,866	7,863	6,589
	#3	10,000	6,129	4,169	4,197	2,110
RLE	#1	10,000	8,235	5,096	7,810	2,346
	#2	10,000	7,954	6,079	7,617	6,792
	#3	10,000	8,160	3,732	3,942	1,173
MF	#1	10,000	9,670	9,290	8,757	5,504
	#2	10,000	9,457	7,523	8,337	6,614
	#3	10,000	9,720	5,139	5,719	3,165

With a unlimited sized CFC-ST able to store signatures of the BBs, the test capability is reported in Table 2.3 using the percentage of faults detected out of the faults causing the CFEs and Wrong Results as metric.

Furthermore to evaluate firstly the techniques for choosing the BBs whose signatures to be pre-computed and stored for the static CFC-ST when the size of CFC-ST is smaller than the number of BBs in the software, fault injection was carried out with the same setup with different BB ranking techniques described before applied. The fault injection result in Table 2.4, however, with only the larger ones in all the benchmark applications, shows that the Fault Coverage, although clearly decreasing with the CFC-ST size, does not collapse, but slowly decreases, allowing the designer to trade-off between achieved

Table 2.3: Detection capabilities (Unlimited CFC-ST size)

	Fault Model	miniMIPS		LEON3	
		Faults Causing CFEs & Wrong Results	Detected (%)	Faults Causing CFEs & Wrong Results	Detected (%)
Bubble	#1	4,145	100	3,122	100
	#2	4,148	100	5,243	100
	#3	2,453	100	623	100
Matrix	#1	6,079	100	5,232	100
	#2	6,908	100	5,042	100
	#3	3,421	100	4,577	100
Dijkstra	#1	3,069	100	3,683	100
	#2	3,866	100	6,589	100
	#3	4,169	100	2,110	100
RLE	#1	5,096	100	2,346	100
	#2	6,079	100	6,792	100
	#3	3,732	100	1,173	100
MF	#1	9,290	100	5,504	100
	#2	7,523	100	6,614	100
	#3	5,139	100	3,165	100

Fault Coverage and hardware cost. Clearly, the method effectiveness weakens when the size of the program overcomes a given threshold, as it happens for the largest among our benchmarks.

Even though the Static-CFC module can provide high fault coverage with relatively low cost when the number of BBs in the application to be monitored is low, when it comes to large software running on the processor, the Static-CFC method will lose information of most of BBs and fail in achieving an acceptable fault coverage. The Dyn-CFC module can be used in these cases. Table 2.5 reports fault injection results with the same larger benchmark applications.

As can be seen from the figures, the Dyn-CFC module can still achieve very high fault coverage, even when the CFC-ST size is significantly smaller than the number of BBs to be monitored. The fault coverage appears to be lower for smaller benchmarks than for larger ones. The reason for this behavior is because the Dyn-CFC method starts with an empty CFC-ST, and hence cannot detect some faults in the early phases of the application execution, which can be detected by the static method. After the CFC-ST is filled up, the dynamic method becomes more effective than the static one. For large applications the initial drawback is negligible, and the advantages of the dynamic method (in terms of ability to dynamically store in the CFC-ST the most useful information) dominate. In this scenario, the designer need to determine which version and which size of the CFC-ST is most suitable for the actual system to achieve optimal results.

A new method was proposed in this work for online test of Control Flow Errors by

Table 2.4: Fault coverage with a smaller static CFC-ST

	CFC-ST Size	Fault Model	Detected (miniMIPS)			Detected (LEON3)		
			T1	T2	T3	T1	T2	T3
			%	%	%	%	%	%
Dijkstra	16	#1	89.35	98.11	98.11	97.80	94.46	99.24
		#2	91.33	96.61	96.43	99.94	99.94	99.94
		#3	83.71	98.22	98.22	98.48	98.53	98.53
	8	#1	71.91	75.86	88.95	43.36	77.36	71.63
		#2	63.32	79.51	88.57	99.94	99.94	99.94
		#3	63.49	87.57	89.66	83.84	95.36	91.85
RLE	32	#1	85.69	98.00	98.00	71.44	99.66	99.66
		#2	92.45	96.46	97.07	100	100	100
		#3	88.83	96.60	96.60	90.20	99.06	99.06
	16	#1	81.30	77.53	89.60	46.80	58.65	62.66
		#2	61.23	69.11	85.08	100	100	100
		#3	83.31	60.26	84.38	83.80	91.82	88.92
MF	128	#1	99.09	99.89	99.93	97.52	99.80	99.83
		#2	69.03	99.74	99.75	99.93	99.98	99.98
		#3	39.97	99.65	99.82	87.48	99.80	99.98
	64	#1	97.96	98.73	99.22	87.97	98.77	98.92
		#2	60.45	97.69	98.10	99.93	99.98	99.98
		#3	24.91	95.74	96.69	80.36	99.12	99.32
	32	#1	96.54	96.43	96.51	43.42	98.23	98.23
		#2	53.70	93.45	94.66	99.90	99.98	99.98
		#3	21.29	90.19	91.17	67.58	98.83	98.91
	16	#1	95.94	95.47	95.34	42.85	93.59	93.87
		#2	52.73	89.15	89.96	99.90	99.98	99.98
		#3	7.90	86.32	85.02	67.22	95.08	95.24

exploiting the debug infrastructure that already exists in some processors in market. The information which can be extracted on the fly from the processor using the debug interface are managed by an external module which is able to perform a few checks and to guarantee a high Fault Coverage with respect to CFEs.

The method does not require any change either in the processor hardware or in the application software: it only involves adding outside the processor an application-independent module which monitors the output of the debug port, and compares it with the expected values. With the static CFC module, signatures of the BBs need to be statically computed at compile time, or when the machine code is available, while Dyn-CFC can be used directly without such requirement. No performance penalty is involved. A major advantage of the proposed method with respect to previous ones lies in its ability to work even with processors equipped with caches.

Table 2.5: Detection capabilities with Dyn-CFC module

	CFC-ST Size	Fault Model	miniMIPS	LEON3
			Detected (%)	Detected (%)
Dijkstra	16	#1	86.02	93.76
		#2	98.89	99.94
		#3	98.22	95.36
RLE	32	#1	96.29	97.53
		#2	98.83	100.00
		#3	98.04	99.66
	16	#1	92.50	89.60
		#2	81.74	100.00
		#3	89.34	95.82
MF	128	#1	99.78	99.92
		#2	99.70	99.98
		#3	99.32	99.96
	64	#1	99.49	99.49
		#2	99.36	99.98
		#3	98.74	99.76
	32	#1	98.41	98.97
		#2	97.64	99.98
		#3	97.14	99.39
	16	#1	96.26	97.89
		#2	94.96	99.98
		#3	93.54	98.14

Depending on the actual application to be monitored, designer can choose which version of the CFC module should be applied, and the size of CFC-ST to achieve reasonable test coverage taking into considering the resource overhead of the CFC module. In case, the static CFC-ST is applied, designer can use the BB ranking techniques proposed in this work to cope with a size-limited CFC-ST.

Experimental results gathered by performing extensive fault injection campaigns on the miniMIPS and LEON3 processors show that the method detects a high percentage of the faults causing Control Flow Errors. Although the resulting figures are sometimes slightly lower than the ones achieved by other methods, it should be taken into account that this result comes at practically no cost in terms of hardware or performance overhead. Experimental results also demonstrate that the hardware overhead of the proposed external module is very limited.

As a conclusion, the method proposed (that can be complemented by other methods to address data faults, too) provides high fault detection capabilities at low cost, and its adoption is characterized by minimal intrusiveness and easy integration into existing design flows.

Chapter 3

Hybrid Nonintrusive Error Detection Technique

As discussed in previous chapter, the CFC module proposed can be used for online CFE detection, and to handle Data Errors, it needs to be complied with other techniques for Data Error detection. So this chapter discusses a new hybrid nonintrusive error detection technique using dual control-flow monitoring[1]. The hybrid technique involves an external hardware module and a combined software-based data hardening technique applied at high level source code.

This chapter is organized as follows: firstly, information related to Single Event Effects causing CFE and Data Errors is provided in the background section; then the mechanism implemented in the hardware module for dual Control-Flow monitoring is explained, followed by the data hardening technique applied for handling data errors; finally the fault injection campaign results and analysis is presented, proving that the proposed hybrid technique is able to detect a high percentage of soft errors caused by Single Event Effects in processor-based systems.

3.1 Background

Nondestructive *Single Event Effects* (SEEs), also known as soft errors, are an increasing concern for the reliability of complex digital systems. They occur when a particle strikes a node in a circuit and generates a transient voltage pulse that can propagate within the circuit [19]. When the transient pulse occurs in a memory element, such as a register, it is known as *Single Event Update* (SEU). When the transient pulse occurs in a combinational element, the effect is known as a *Single Event Transient* (SET).

Errors produced by SEEs in a microprocessor are usually divided into *Data Errors* and *Control Flow Errors*. If an error occurs in a register or memory position storing data, a

[1]This work was done with collaboration with our colleagues in University Carlos III of Madrid, Leganes, Madrid and supported in part by EU FP7 STREP project BASTION; related publications can be found as [52]

wrong computation result may be obtained. If an error occurs in a control register, such as the program counter or the instruction register, the instruction flow may be corrupted and a wrong result may be produced or the processor may lose control and enter an infinite erroneous loop. Both types of errors can be detected using software techniques. Fault-tolerance techniques based on software rely on adding extra instructions to the original program code to detect or correct faults [49]. Software-based techniques provide high flexibility, low development time and low cost, since they can be implemented without modifying the hardware. However, software-based techniques cannot achieve full system protection against soft errors [9] and may produce large overheads in processing time and storage needs, particularly when designed to protect the microprocessor against CFEs [7, 6].

Hybrid techniques involve combining hardware and software fault-tolerant techniques to improve error detection and reduce the performance degradation that software techniques entail. The hardware technique typically consists of introducing an external hardware module to monitor the execution of instructions in the processor. With the hybrid technique discussed in this chapter, both the input and output of the instruction stream, which are captured through system bus between memory and processor and trace interface of the processor respectively, are monitored. A major advantage of the proposed approach is that it does not require any software modification and, therefore, it produces no performance degradation. By complementing it with software fault-tolerance techniques to cover data errors, a complete solution against SEEs with reduced performance degradation and low memory overhead is obtained.

3.2 Dual Control-Flow monitoring

In a processor-based system, the processor retrieves instructions to be executed from memory and the trace interface in the processor (such as the one in LEON3) is able to provide the execution flow of the instructions executed by the processor. The input and output of the instruction stream are monitored by the the external hardware module, and together with the data hardening technique, this approach can achieve high fault coverage.

3.2.1 External hardware module

The external hardware module is used to monitor the instructions executed by the processor both at the input and output stream. A system adopting the hardware monitor is shown in Fig. 3.1.

One of the observation points of the hardware monitor is the memory or cache bus (input of the instruction stream). This bus provides information of the program counter (PC) and the instruction code (opcode and operands) at the fetch stage, just when the instruction is loaded in the microprocessor.

The second observation point is the instruction trace interface (debug interface as previous chapter), which provides the most relevant information of each executed instruction, including the PC, instruction code (opcode and operands), time tag and trap and error flags [28]. This information is provided just after the instruction is executed. In the case

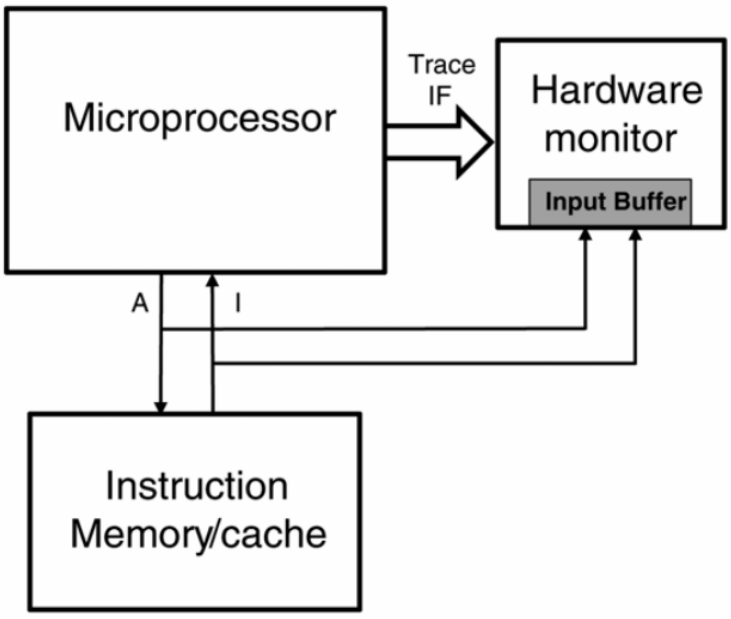


Figure 3.1: Hardware monitor observation points

of LEON3, which has seven pipeline stages (fetch, decode, register access, execute, memory, exception, and write back), the information in the trace interface corresponds to the exception stage. The trace interface can be accessed without affecting normal operation of the processor or adding any performance penalties. Moreover, the use of the trace interface as an observation point does not interfere with the possible use of the trace interface for debugging purposes.

Once an instruction is loaded from the memory, the instruction information travels along the microprocessor data path and is used in each stage to drive the operation of the microprocessor. An error which occurs in the PC or the instruction register (IR) at any stage will be finally observed at the trace interface and can be detected by comparing the trace interface output with the upstream information collected at the fetch stage. Notwithstanding, an error in the PC at the fetch stage may not be detected because it is issued by the microprocessor. When such an error occurs, both observation points (memory bus and trace interface) provide the very same information, but it is erroneous. To improve the detection capabilities including those errors, a PC prediction technique is used [54].

PC prediction is a control-flow checking approach that consists of predicting the next PC value by checking the opcode and the present PC value. The predicted PC value is then compared with the PC value of the next executed instruction. If there is any difference between both PCs, an error in the program flow is detected. The opcode of every new instruction executed is checked. If the opcode corresponds with a branch instruction, the PC must be incremented either by the branch offset if the branch is taken, or by the

instruction size if the branch is not taken. For a nonbranch instruction, the PC must be incremented by the instruction size. The trace interface provides the information required by the PC prediction technique just after the instruction is executed.

A dedicated hardware monitor (HM) module has been designed to implement the proposed approach. Fig. 3.2 shows the internal structure of the HM. There is an interface for each observation point and a block for each implemented technique. The control block is responsible for the correct behavior of the different blocks and interfaces.

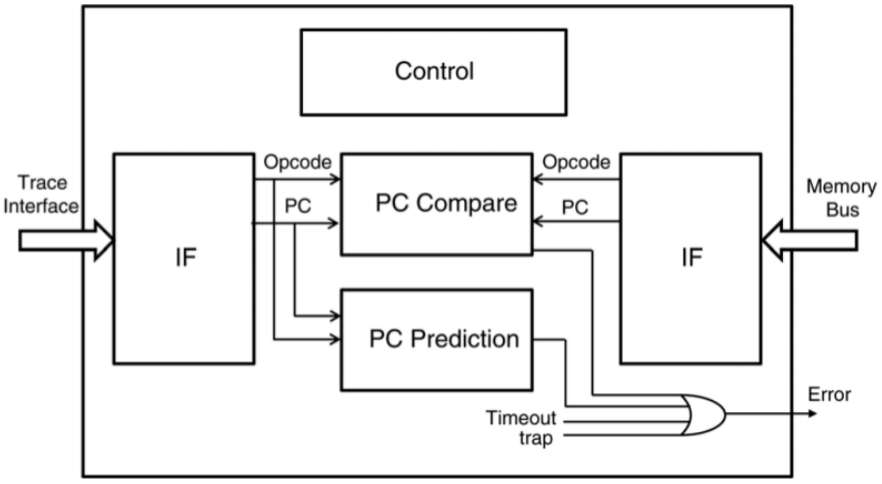


Figure 3.2: Internal architecture of the hardware monitor

The information provided at the two observation points must be synchronized for a correct comparison. For this purpose, upstream data are stored in an input buffer with a size equal to the number of pipeline stages of the processor. When a new instruction appears in the memory bus at the fetch stage, the HM catches the opcode and the PC of the instruction and stores them in the input buffer. Then, every new instruction provided by the trace interface is compared with the instruction stored in the input buffer in order of appearance. It must be noted that the error detection latency is minimal, because an error can be detected as soon as the executed instruction appears at the trace interface.

In addition to instruction comparison and PC prediction, the control module of the hardware monitor also checks the time tag and the trap and error flags provided in the trace interface. The trace interface time tag is the output of an internal counter inside the processor that is incremented in each clock cycle as long as the processor is running. A timeout condition is set to cope with the case the processor hangs in a particular instruction. The timeout condition activates the error signal if the time tag advances without issuing new instructions for a long period of time.

The trap and error flags provided by the trace interface are used for exception handling. To implement exception handling, it is important to differentiate between fault-induced exceptions, which may be caused by an SEE, and implemented exceptions, which are expected to occur under normal execution. The HM uses the trace interface flags to

detect fault-induced exceptions that cause an unexpected trap or the processor entering error mode. Unexpected traps can be caused in several ways, such as invalid instructions or invalid memory addresses. They can be differentiated from implemented traps by checking the next instruction provided by the trace interface. The trap signal of Fig. 3.2 is triggered when an unexpected trap occurs or the processor enters error mode.

The HM can be implemented with small hardware overhead since it does not require storing information obtained at compilation time. The proposed module can work with the observed information without disturbing the normal microprocessor behavior. Moreover, the HM can detect control-flow errors without any specific support from the application software.

3.2.2 Data hardening technique

A software-based data hardening to lessen the impact of data-flow hardening in the HM. Hardware-based data-flow hardening requires additional connections to the microprocessor architecture that are not easily available. Another drawback of the hardware implementations is the area increase. Data hardening techniques require instruction re-execution with the corresponding additional storage.

A combination of two software-based technique, namely total data-flow duplication and inverted branches, was used to cover Data Errors.

Total data-flow duplication duplicates all of the software data and compares both data flows whenever a write operation is performed. When a discrepancy between the two data flows appears, an error is detected. This method achieves good data error coverage but increments code size and decreases performance. In order to reduce the performance penalty as well as the code size, the checking points of the code were reduced as proposed by [50]. In [50], the entire data flow is duplicated but only certain variables (final variables) are checked. A similar approach can be found in [8] where the number of checks is varied depending on system requirements. This approach maintains the data integrity since every operation is performed twice but checking instructions are considerably reduced, at the expense of some acceptable increase in error detection latency, that is, the time between an error occurs and when it is detected.

Another software hardening technique has been applied to our code in order to detect errors in conditional branches. In a conditional branch, errors may appear in the evaluation of the condition codes or in the condition codes themselves, resulting in the branch being incorrectly taken or not taken. The technique called "inverted branches" [8] was used to detect errors in conditional branches. This technique re-evaluates the branch condition in two locations. When the branch is taken, the branch instruction is repeated with an inverted condition. In addition, when the branch is not taken, the branch instruction is simply repeated. The objective of this technique is to repeat the evaluation of the condition codes. If the repeated evaluation does not produce the same result, an error is detected.

3.3 Fault injection campaign

A fault injection campaign targeted on LEON3 processor was carried out to validate our approach. A basic configuration has been built, including one LEON3 integer unit with eight register windows, instruction and data caches (2 kB each), instruction trace interface, interrupt controller, system bus (AMBA), memory controller, and general purpose input/output. The memory controller can drive external random-access memory (RAM) and read-only memory (ROM) where code and data are stored. It must be noted that this system includes several components, besides the LEON3 processor core, that are typically needed to interface the processor. Hardening these components is beyond the scope of this work. However, they have been kept in the system because, in practice, it is very difficult to clearly distinguish them from the LEON3 processor hardware.

The HM logic area is about 22% of the LEON3 logic area, excluding the memories that implement the register file. Table 3.1 shows the synthesis results for 90-nm technology. It must be noted that the utilized LEON3 configuration is minimal. Upgrading LEON3 with additional modules does not modify the HM architecture and does not increase the HM area.

Table 3.1: Synthesis results

	#Gates	#FFs	Area(um^2)	Memory
LEON3	7,185	1,851	116,881	16Kb
HM	1,230	399	27,613	512b

Three software applications have been used for testing:

1. *BBS* implements the bubble sort algorithm for a vector of 15 values,
2. *Mmult* implements a 5x5 matrix multiplication,
3. *AES* implements the AES encryption algorithm.

In all cases, intermediate computation results are frequently sent to a parallel output port, where they can be checked during the fault-injection process. All algorithms were developed in C and compiled with GCC using the -O2 optimization option. To better demonstrate the capabilities of our approach, the experiments were first conducted with an unhardened version of the application software, as given by the compiler with no further manipulation to harden it. Then, the experiments were repeated with a software version that is hardened for data errors as described in previous section. The hardened software version was also developed in C and compiled with the same options.

In the experiments, the same approach as in [42] and [53] was adopted to evaluate the error detection capabilities. We estimate the global error rate using fault injection. The dynamic cross-section can then be calculated as the product of the static cross-section and the estimated global error rate. Because the static cross-section is the same for the hardened and unhardened versions of the circuit, relative comparisons can be made in

terms of the global error rate. Moreover, fault injection allows us to perform a more detailed error analysis.

To obtain the global error rate, the AMUSE tool [24, 25] was used. This tool is an emulation-based fault-injection system that can cover SEU and SET, including logical, latch-window, and electrical masking effects. It also provides very high performance, which enables very large fault-injection campaigns to be executed in a short time. With respect to test coverage, as described in [58], AMUSE typically provides 100% coverage of expected radiation test results with respect to fault locations, input vectors, and clock cycles of operation for small- or medium-size test cases.

Fault-injection campaigns were conducted for SEUs and SETs. For SEU experiments, SEUs were injected in every flip-flop and clock cycle, covering the full SEU space of the application. For SET experiments, faults were injected at several random instants within every clock cycle for every gate and with a pulsewidth of 10th of the clock period, using the approach described in [25].

In the experiments, errors were classified in several categories, following the terminology proposed in [46]. Errors that are not detected by either the HM or hardened software are classified as silent data corruption (SDC) or Hang. An error is classified as SDC as soon as an erroneous output is observed at the output port. An error is classified as Hang if no new values are observed at the output port for a long time, which indicates the processor may be lost. For this purpose, a timeout condition has been established with some extra clock cycles that enable correct completion of the computation. An error is classified as Hang if the timeout condition is overtaken. Note that a Hang error can be produced by control-flow error (e.g., an incorrect jump) or by a data error (e.g., an error in the index of a loop that prevents the program from finishing in due time).

Tables 3.2-3.4 summarizes the results of the fault-injection campaigns with the HM for the three selected software applications with unhardened and hardened software versions. The internal registers of the LEON3 have been divided in two sets: Set I includes the PC & IR for all stages (346 FFs) and Set II includes the remaining registers (1,505 FFs). The first three rows in the tables show the results of SEU fault injection for sets I and II, and all of the registers, respectively. The last row shows the results of SET fault injection. From left to right, the table shows the number of injected faults, the total amount of observed errors, and the classification of errors as SDC, Hang, or Detected by the HM. The percentage of errors in each category with respect to the total amount of observed errors is provided in brackets.

Errors reported in the table are true errors, that is errors that produce wrong observable behavior. False errors, such as those can occur in the hardware monitor, have not been included. The effect of a false error is to trigger unnecessary error-recovery action. For low error rates, the impact of some sporadic error-recovery action is negligible. Otherwise, the hardware module can be hardened to reduce the chance of false errors.

As shown in Table 3.2, the unhardened Bubble Sort algorithm takes 3404 clock cycles. Therefore, 3404 SEUs were injected per flip-flop, up to 6.3 million SEUs in total. 10234 SETs were injected per combinational node, up to 80.6 million SETs in total. Taking into account the large amount of injected faults, the error margin is smaller than 0.1% with 95% confidence [72].

Table 3.2: Fault injection results with BBS

Benchmark	Elements	Faults injected	Errors observed	SDC	Hang	Errors detected
BBS, Unhardened SW	PC & IR(I)	1.177M	343,278	0	0	343,278 (100%)
	Other Regs(II)	5.123M	361,307	167,192 (46.3%)	36,764 (10.2%)	157,351 (43.6%)
	All Regs	6.301M	704,585	167,192 (23.7%)	36,764 (5.2%)	500,629 (71.1%)
	Comb. logic(SETs)	80.649M	777,634	258,768 (33.3%)	24,579 (3.2%)	494,287 (63.6%)
BBS, Hardened SW	PC & IR(I)	2.997M	767,712	0	0	767,712 (100%)
	Other Regs(II)	13.038M	813,107	81,996 (10.1%)	45,195 (5.6%)	685,916 (84.4%)
	All Regs	16.035M	1,580,819	81,996 (5.2%)	45,195 (2.9%)	1,453,628 (92.0%)
	Comb. logic(SETs)	205.233M	1,881,373	78,992 (4.2%)	10,448 (0.6%)	1,791,933 (95.2%)

Table 3.3: Fault injection results with Mmult

Benchmarks	Elements	Faults injected	Errors observed	SDC	Hang	Errors detected
Mmult, Unhardened SW	PC & IR(I)	2.125M	466,416	0	0	466,416 (100%)
	Other Regs(II)	9.245M	599,949	264,051 (44.0%)	50,084 (8.3%)	285,814 (47.6%)
	All Regs	11.371M	1,066,365	264,051 (24.8%)	50,084 (4.7%)	752,230 (70.5%)
	Comb. logic(SETs)	145.533M	1,495,468	436,119 (29.2%)	25,499 (1.7%)	1,033,850 (69.1%)
Mmult, Hardened SW	PC & IR(I)	5.11M	1,173,644	0	0	1,173,328 (100%)
	Other Regs(II)	22.3M	1,586,301	65,174 (4.1%)	117,046 (7.4%)	1,404,397 (88.5%)
	All Regs	27.4M	2,759,945	65,174 (2.4%)	117,046 (4.2%)	2,577,725 (93.4%)
	Comb. logic(SETs)	350.3M	3,921,304	69,807 (1.8%)	50,633 (1.3%)	3,800,864 (96.9%)

Table 3.4: Fault injection results with AES

Benchmarks	Elements	Faults injected	Errors observed	SDC	Hang	Errors detected
AES, Unhardened SW	PC & IR(I)	1.514M	833,840	0	0	833,840 (100%)
	Other Regs(II)	6.587M	610,326	321,583 (52.7%)	11,968 (2.0%)	276,775 (45.31%)
	All Regs	8.102M	1,444,166	321.583 (22.3%)	11,968 (0.8%)	1,110,615 (76.9%)
	Comb. logic(SETs)	103.701M	1,017,164	324,243 (31.9%)	1,439 (0.1%)	691,482 (68.0%)
AES, Hardened SW	PC & IR(I)	2.271M	1,150,973	0	0	1,150,973 (100%)
	Other Regs(II)	9.879M	762,482	107,390 (14.1%)	30,487 (4.0%)	624,605 (81.9%)
	All Regs	12.150M	1,913,455	107,390 (5.6%)	30,487 (1.6%)	1,775,578 (92.8%)
	Comb. logic(SETs)	155.511M	1,336,025	78,167 (5.9%)	2,999 (0.2%)	1,254,859 (93.9%)

The proposed approach is able to detect 100% of the errors in Set I and many of the errors in Set II. Although *Set I* is much smaller than *Set II*, it accounts for about half of the total observed errors. This is because the PC and IR registers are very critical. In particular, Set I accounts for all control-flow errors [76]. Errors in other registers (Set II) may produce a wide variety of effects, but they can also be detected by the HM if they eventually produce a control-flow error, invalid addresses, infinite loops, etc. The HM is also able to detect a similar percentage of errors caused by SETs.

When the *BBS* application is hardened with SW technique, it takes 8,663 clock cycles, and the amount of injected faults goes up to 16 million SEUs and 205 million SETs. Again, all errors in Set I are detected. Some of these errors may be detected by software, if the software error detection triggers earlier than the HM. By combining the HM with software hardening for data errors, 92.0% of SEUs and 95.2% of SETs are detected. The majority of the remaining undetected errors correspond to faults injected outside the processor core, which are not covered by the proposed approach. For instance, an error in the memory controller or the bus controller may be affected in a common way to duplicated variables and, therefore, may not be detected by the hardened software. Protection against these errors should be provided by other means, which are outside the scope of this work.

The *Mmult* application is more complex and requires 6,143 clock cycles to complete for the unhardened software case, and 14,788 clock cycles for the hardened software case. Therefore, the amount of injected faults is increased to provide the same test coverage. The error detection capabilities are very similar to the BBS application. Again, the HM detects all errors in Set I and many of the errors in Set II. For this software application,

93.4% of SEUs and 96.9% of SETs were detected with a combination of the HM and software hardening for data errors.

The *AES* application has a larger code, although it executes in fewer clock cycles, namely, 4,377 clock cycles using unhardened software and 6,564 clock cycles for the hardened software version. The error detection capabilities are again similar to the other applications.

Finally, several fault-injection campaigns were performed on the register file. The purpose of these campaigns is to evaluate the capability of the HM for error detection, even though errors in the register file are data errors. The register file of LEON3 consists of two RAM modules that implement the eight register windows and the eight global registers. The RAM modules are commonly protected by using radiation-hardened memory or using error detection and correction (EDAC) codes. Otherwise, software fault-tolerance techniques can be used. For these campaigns, assumption has been made that the RAM modules are not protected except for the error detection mechanisms of the HM and the implemented hardened software.

The results of the fault-injection campaigns on the register files for several software applications are summarized in Table 3.5. Again, the full SEU space was covered and SEUs were injected in every RAM bit and clock cycle. The first three rows show the results for the BBS, Mmult, and AES applications, respectively, using the HM with unhardened software. Although the injected faults produce data errors, the HM is able to detect 30.8%, 42.8%, and 38.0% of them, respectively. These errors are mainly detected by the timeout and exception handling features of the HM. The next three rows in Table 3.5 show the results using hardened software versions. In these cases, the error detection rate rises to 93.3%, 92.4%, and 97.3%, respectively. The reason why no full error detection is achieved is that the compiler optimizes away some of the redundant code used for error detection. Error detection can be improved by reducing the compiler optimization level at the expense of increasing the execution time. For instance, the error detection rate in the Mmult application rises to 99.3% by reducing the compiler optimization level to , as shown in the last row of Table 3.5.

The novel hybrid approach proposed in this work for error detection in microprocessors which is based on monitoring and comparing the instruction flow at the input and at the output of the microprocessor. The proposed technique is intended for complex microprocessors with several pipeline stages in which instructions can be corrupted as they move into the pipeline of the processor. This technique has several advantages with respect to previous approaches that use a single observation point. First, it does not require software modifications or additional information to compare with. Second, since the control flow is observed at two different points, just before and after instruction execution, it can detect any error that occurs in between.

Experimental results with the LEON3 microprocessor demonstrate that the proposed approach can achieve 100% control-flow error detection. On the other hand, CFEs account for the majority of errors. By complementing it with software-based fault-tolerance techniques, which are only required for protection against data errors, a complete solution against SEEs with reduced performance degradation and low memory overhead can be obtained.

Table 3.5: Fault injection results with Register File

Benchmarks	Faults injected	Errors observed	SDC	Hang	Errors detected
BBS (HM)	29.628M	509,955	310,197 (60.8%)	42,467 (8.3%)	157,291 (30.8%)
Mmult (HM)	53.469M	1,413,124	580,597 (41.1%)	227,806 (16.1%)	604,721 (42.8%)
AES (HM)	71.708M	1,414,901	844,203 (59.7%)	32,470 (2.3%)	538,228 (38.0%)
BBS (HM+SW)	75.403M	2,626,626	175,623 (6.7%)	0 (0%)	2,451,003 (93.3%)
Mmult (HM+SW)	128.715M	1,510,704	49,016 (3.2%)	65,885 (4.4%)	1,395,803 (92.4%)
AES (HM+SW)	107.538M	3,479,705	22,668 (0.7%)	72,626 (2.1%)	3,384,411 (97.3%)
Mmult (HM+SW -O0)	284.621M	1,976,912	7,864 (0.4%)	6,040 (0.3%)	1,963,008 (99.3%)

Chapter 4

Printed Circuit Board Assembly Power-On Self-Test

This chapter describes the method to exploit the processors' debug features for *PCBA Power-On Self-Test* (POST)[1]. As discussed in previous chapters, the debug interface is able to provide information regarding the execution of software on the processor. In processor, such as LEON3, a stream of executed instruction along with corresponding Program Counter value can be retrieved through the debug interface; and in processor, such as ARM, even though no such stream is generated, but the CoreSight components are still able to generate rich useful data including target address of an executed branch instruction, exception information and clock cycle count between two branch instructions etc., which can contribute to improve processor observability during POST. And the proposed technique takes advantage of such information, introducing a Monitoring IP can be mapped to a FPGA device on the same board.

This chapter is organized as follows: firstly, information regarding POST and functional test is provided in the background section; and then an introduction of the CoreSight architecture provided by ARM is presented, explaining the possible information regarding the software running on the processor can be extracted through the debug/trace interface; afterward, the architecture of the proposed Monitoring IP with the workflow how this technique can be adopted is explained proving the feasibility for implementing such a component; finally, due to lack of internal knowledge of the ARM processor, experiment result with miniMIPS processor is presented, with comparison between the fault coverage of the proposed technique and when all outputs of the processor is observable, proving the effectiveness of the approach with Monitoring IP.

[1]This work was done with collaboration with our colleagues in Universidad de la República Montevideo, Uruguay, and Testonica Lab, Tallinn, Estonia and supported in part by EU FP7 STREP project BASTION; As the time of writing, this work has been accepted as a regular paper in 2016 IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems

4.1 Background

Power-On Self-Test (POST) plays an important role in many systems, since it may detect faults arising during the life time of the product, thus increasing its dependability. POST may use different solutions, which should match the constraints of the environment the system is deployed in.

Functional test [67, 65] represents a commonly adopted solution for POST. More in general, functional test is adopted in many scenarios, at the device, board and system levels [39]. In some of them it complements other test steps, performed resorting to *Design For Testability* (DFT). For example, when considering Printed Circuit Board Assemblies (PCBAs) test, it is common to see functional test as the last step, mainly targeting dynamic defects. The importance of this kind of defects is significantly increasing in the last years, especially since they are considered one of the major contributors for *Non Failure Found* (NFF) [37, 41], thus raising the interest for any solution able to improve the achievable defect coverage.

Hence, functional test plays a key role for in-field test of PCBAs [26]. In this context, functional test is particularly attractive due to its relatively low implementation cost (no equipment required), low intrusiveness (no or limited changes in the PCBA design and low impact on the application) and flexibility (since functional test is typically based on forcing the processor to execute a suitable test program, possibly mimicking some specific application code). A commonly adopted solution consists in launching at the power-on the execution of a functional test targeting all the critical modules in the system, or, alternatively, some specifically crafted test able to make observable the highest percentage of defects [55]. Functional test is normally based on a sequence of instructions to be executed by the in-system processor(s), aimed at exciting possible defects in the processor itself or in any other device on the board, as well as in the interconnect. Therefore, this kind of test is sometimes referred to as Software-Based Self-Test, or SBST [57].

Major limitations to the effectiveness of functional test include

- The difficulty in assessing the achieved defect coverage: although many efforts have been done to introduce high-level metrics, their correlation with real defect coverage is still a matter of discussion and a hot research topic [36].
- The cost for creating suitable functional stimuli; a lot of work has been done to automate this part, or at least to provide guidelines for the test engineer in charge of developing suitable functional tests.
- The limited observability that can be obtained on the behavior of the system under test, both as a whole and in terms of its components.

The method discussed in this chapter focuses on the last point, and is an approach based on combining different solutions.

First of all, the method uses the features currently provided by many processors in order to support the debug of the software for test purposes. In particular, these features often allow to access on-the-fly during the normal behavior of the processor (and without slowing it down) to several information about its internal behavior. As an example, they

typically allow to trace the sequence of instructions executed by the processor, writing them to ad hoc external interfaces.

Secondly, since the on-the-fly monitoring of the flow of data produced by the debug interface can only be done resorting to ad hoc hardware, a module was proposed to be mapped on an FPGA, assuming that some programmable hardware is available on the board [38].

Finally, a scheme was described, in which the access to the debug features supported by the module mapped on the FPGA is integrated into a board-level test environment that can be used for in-field test.

In order to assess the effectiveness and limitations of the proposed solution, a Zynq-7000 system by Xilinx was used, which integrates one or more ARM processors and a programmable module. An IP core was developed that can be mapped on the latter and is able to monitor the TPIU (Trace Port Interface Unit) debug interface provided by ARM. Also a small software library was developed to be integrated into the code of the functional test, so that the debug features are properly programmed at the beginning of the test, and then used to increase the observability during the test execution. In this way, the feasibility of the proposed solution and its (rather limited) cost in terms of required FPGA resources could both be demonstrated.

Finally, since the lack of detailed information about the structure of the ARM processor clearly prevents us from computing the increase in defect coverage that can be achieved using the solution, some fault simulation experiments were performed on a MIPS-like processor for which the model is available. The same information produced by the ARM debug interface is extracted from this processor during the execution of the test, and the achieved fault coverage is computed. Results show the effectiveness of the proposed solution. Interestingly, they demonstrate that the stuck-at fault coverage that can be achieved is comparable with the one reachable using a corresponding test program in a scenario where all the processor outputs can be continuously monitored.

4.1.1 CoreSight Architecture from ARM

A Zynq-7000 SoC by Xilinx was used as a platform for concepts evaluation. The Zynq device has a ARM-based SoC with a CortexA9 dual core processor embedded in the same chip with the FPGA part, and the ARM-based SoC is equipped with the CoreSight On-chip Trace and Debug Architecture [4] that can output trace data via TPIU from which custom IP mapped on the FPGA part retrieve data.

According to the CoreSight Architecture, all the CoreSight components belong to one of the following four classes:

1. *Access and Control* includes the *ARM Debug Access Port* (DAP) which provides access to the internal memory and memory-mapped configuration registers of the device controllers (including the CoreSight components) via the JTAG port; the *Embedded Cross Trigger* (ECT) components which implements a mechanism to pass debug events between processor cores (hard and soft cores in case of the Zynq-7000 SoC) allowing one CoreSight component to communicate with others via trigger events.

2. *Trace source* includes components that generate trace information from different aspects of the system. The *Program Trace Macrocell* (PTM) generates trace data for the software running on the processor; the *Instrumentation Trace Macrocell* (ITM) allows software developer to explicitly insert trace points into the software to ease the effort for application-level trace and debug; the *Fabric Trace Monitor* (FTM) is a Xilinx specific trace Macrocell that compiles with the CoreSight architecture specification and enables, in cooperation with ITM and PTM, to trace data generated by peripherals implemented inside the programmable logic (FPGA) of the Zynq-7000 SoC.

3. *Trace link* includes two components for the communication between the trace source and trace sink components. The *Funnel* is in charge of merging trace data from multiple sources (PTM, FTM and ITM) into a single stream and sending it to the *ARM Trace Bus* (ATB); the *Replicator* duplicates the trace stream onto its two output ATM master ports, namely *Embedded Trace Buffer* (ETB) and *Trace Port Interface Unit* (TPIU), from the input ATB slave port.

4. *Trace sink* includes two components for dumping trace information. The ETB is an on-chip storage module with limited size (4KB) which enables short-window real-time and full-speed tracing; while the TPIU allows the trace packages to be output to the FPGA part or to the chip output pins.

The CoreSight components implemented inside Zynq-7000 device is shown in Fig. 4.1.

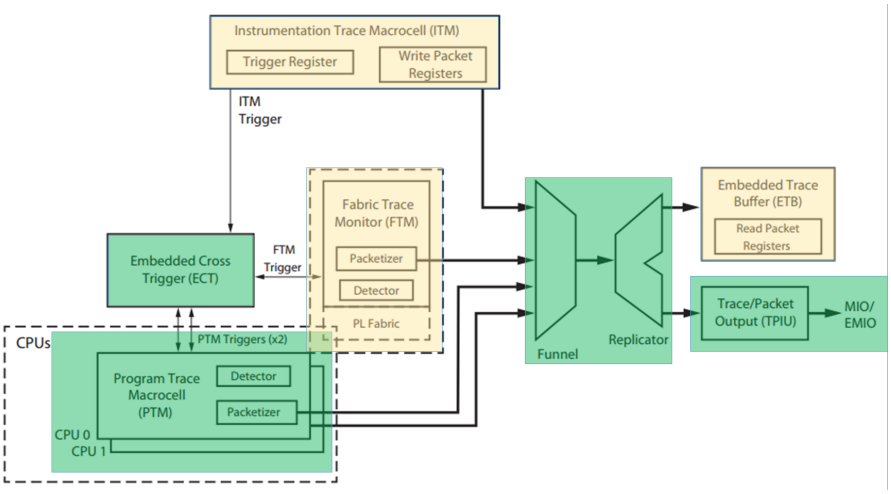


Figure 4.1: CoreSight System Diagram in Zynq-7000[5]

4.2 Monitoring IP

As described above, the CoreSight trace and debug infrastructure implemented in Zynq-7000 allows us to request the processor to generate a flow of trace data and output it

through the TPIU port. In order to monitor this flow of data during the test execution, a custom IP was proposed, which is configured as a peripheral connected to the TPIU and mapped on an FPGA, which is supposed to exist on the board. Hence, following here the general idea already introduced in other papers (e.g., [2]), FPGAs possibly existing on the board for functional purposes can also be used for test purposes. It is important to underline that the proposed approach does not require any additional FPGA resources, since those used for test purpose are then also used for functional purposes when the test is finished.

To demonstrate the feasibility of the approach, a simple monitoring IP was implemented to be mapped on the Zynq-7000 device. With the monitoring IP, it is possible to monitor on-the-fly the trace packages from the PTM and read several relevant information about the test program execution including, for example, taken/not-taken decisions, the target PC address of branch instructions, cycle accurate information between two branch instructions, the exception status of the processor.

To determine if the test running on the processor is executed correctly, the monitoring IP directly compresses the information extracted from the TPIU by sending them to a *Multiple Input Shift Register* (MISR) for signature computation. At the end of the test, the test program itself checks the signature generated by the MISR and compares it with the expected signature.

Note that the Program Flow Trace (PFT) architecture, which is part of the CoreSight specification, defines the types of trace packages as follows [5]:

1. A-sync: (Address synchronization) identifies a package boundary;
2. I-sync: (Instruction synchronization) specifies the processor state for the next instruction to be executed;
3. Atom: Useful when cycle-accurate tracing is enabled;
4. Branch without exception: contains the branch target address;
5. Branch with exception: contains exception vector address;
6. Waypoint update: contains waypoint update address (a waypoint is a point where instruction execution by the processor might involve a change in the program flow);
7. Context ID: indicates Context ID change for the instructions following this package (Context ID indicates source of the packages);
8. Trigger: indicates a trigger occurred (a trigger event can be enabled or defined by user through software);
9. Ignore: has no effect;
10. Exception return: (for ARMv7-M processor) indicates the return from an exception handler;
11. Timestamp: contains the absolute value of timestamp;

12. VMID: indicates the Virtual Machine ID of the processor.

Though the overall set of package types may provide a lot of information about the program flow, only a few of them are worth being used for program flow error checking. Hence, the Monitoring IP does not use (thus simplifying the configuration of the CoreSight components) the packages like Context ID, Timestamp and VMID. Moreover, since a MISR is used to check that the correct sequence of jump instructions is executed, a package filter module is applied to preliminarily remove any information that is non-deterministic (e.g., Timestamp and Ignore packages). The structure of the monitoring IP is shown in the Fig. 4.2 and the resource consumption in terms of Look-Up Tables (LUTs) in the FPGA is reported in Table 4.1.

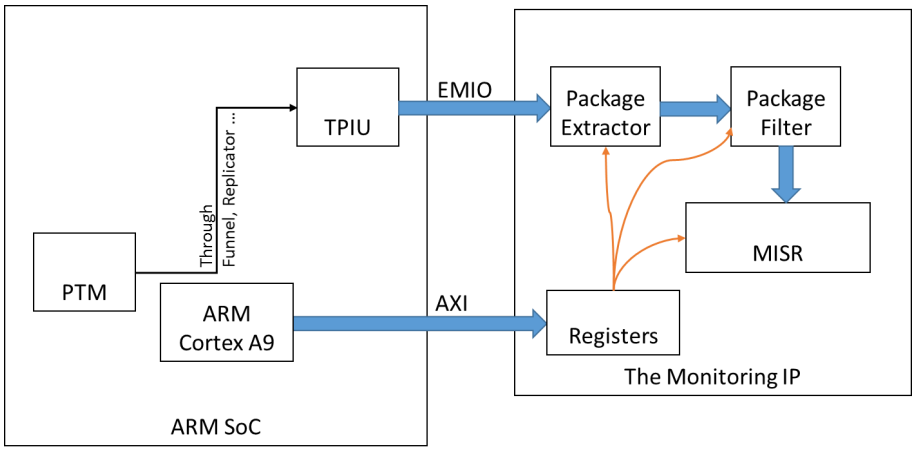


Figure 4.2: Architecture of the Monitoring IP

Table 4.1: Resource consumption for the Monitoring IP

Resource		#Used in Monitoring IP	%Used in Monitoring IP
Slice LUTs	LUTs as Logic	981	5.58
	LUTs as Memory	0	0
Slice Registers	Registers as FlipFlop	1,129	3.21
	Registers as Latch	0	0
Muxes	F7 Muxes	64	0.80
	F8 Muxes	32	0.70

The monitoring IP reads data from the TPIU through the EMIO, and the Finite State Machine (FSM) implemented in the monitoring IP firstly waits for the A-sync package (which is at least 5 bytes of 0) to determine the boundary of the packages. Afterwards, when a header byte is found, the FSM goes into the state devoted to extracting information for the specified package while determining the end of the current package. If not filtered,

the information from the package is then padded, if necessary, and sent to the MISR for calculating the signature.

The FSM uses detailed format about the package to decode and extract information. To provide the reader with an example of the format of the packages, one of the branch address package is described in Fig. 4.3, where only one of the variants depending on the configuration and current processor state is considered. The first section of the package contains 1 to 5 bytes for the address, while the second section contains 0 to 2 bytes for the possible exception information. When the FSM enters the state devoted to extracting information from such a package, it needs to check the bit "C" (see Fig. 4.3) to see if the following byte is still part of the section of the package (i.e., to check the boundary of the package). Then, with the boundary determined, the information extracted (in this case, all the bytes in the package) is sent to the MISR.

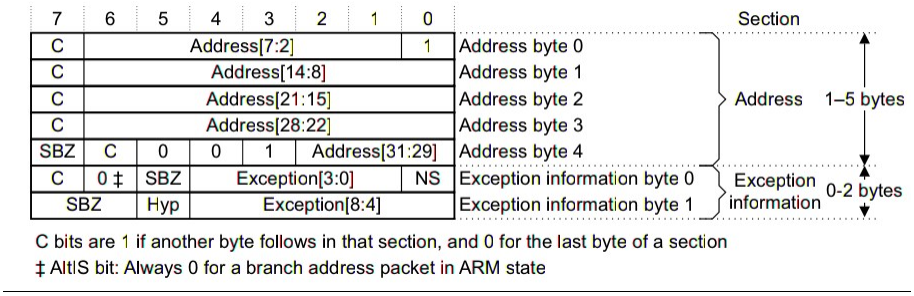


Figure 4.3: Format of the branch with exception package [5]

Furthermore an ARM Advanced eXtensible Interface (AXI) slave was implemented containing several registers which hold the configuration data and run-time status of the other components in the monitoring IP. Through these registers, the user can configure the monitoring IP at the beginning and check the status and final results of the MISR at the end. The whole workflow adopted by the proposed technique is shown in Fig. 4.4.

Following this workflow, the following steps are executed during the Power-On Self-Test of the board

1. The board is powered on, and all the standard initialization procedures are executed
2. The FPGA is programmed with a bitstream stored in some external memory (typically, a Flash); after this step the monitoring IP is available on the FPGA
3. The control passes to the test code (possibly also stored in some Flash memory), which executes the following operations
   (a) Configures the CoreSight components by writing into the proper configuration registers of the device
   (b) Configures the monitoring IP, by writing into its configuration registers
   (c) Launches the execution of the test program; during this phase, each time the test program executes a branch instruction, suitable packets are sent to the

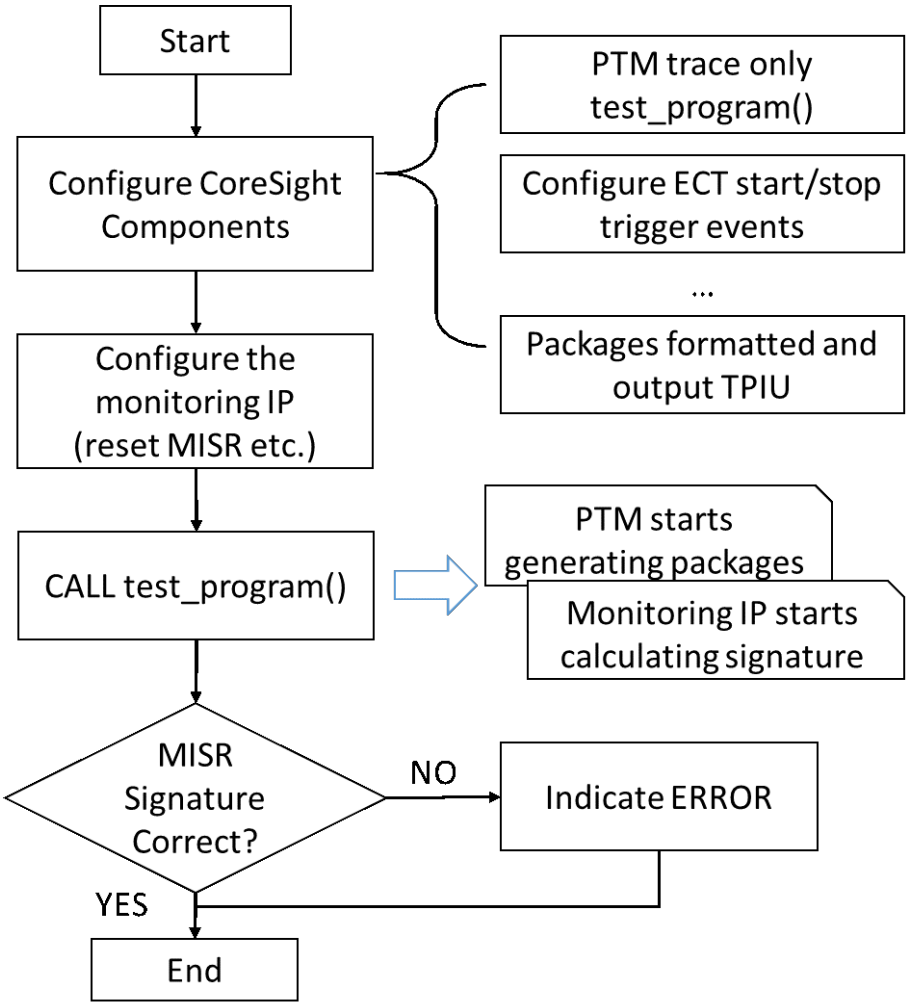


Figure 4.4: Workflow of the proposed technique

TPIU, and processed by the monitoring IP, which extracts the relevant information and compress them in the MISR

(d) Accesses to the monitoring IP, reads the final signature and compares it with the expected one; if a mismatch arises, some error procedure is activated.

Obviously, the test of other parts of the PCBA (including the FPGA itself) is not discussed here, that can be performed resorting to any other mechanism without impacting on the effectiveness of the approach proposed here, which mainly targets the processor module.

4.3 Fault coverage analysis

This section presents some experimental figures aimed at assessing the fault detection increase that one can achieve by adopting the observation mechanism supported by the debug features.

The miniMIPS processor was used, synthesized using Synopsys Design Compiler with a technology library developed in-house. A test program was used, which was manually developed by a test engineer knowing the netlist of the processor and targeting the maximization of the stuck-at fault coverage. The size and duration of this test program are 1,520 bytes and 14,617 clock cycles, respectively. Several fault simulation experiments aimed at assessing the Fault Coverage achievable with such a test program with different observation mechanisms were performed using Synopsys TetraMAX.

Table 4.2 reports for the adopted processor the number of detected faults (out of the total of 266,410 stuck-at faults in the whole processor) using several observation mechanisms:

- Always observing all the output signals of the processor (*ATE*); this mechanism is the one which is typically adopted when performing end-of-manufacturing test of the processor (before mounting it on the board); in this case, the device is mounted on an ATE, which provides full controllability of all input signals, and full observability of all output signals;
- Observing the content of the memory at the end of the test (*mem*); this mechanism is the one which is typically adopted for in-field functional test; the processor executes a test program, and then the final content of the memory area where results are written is checked for compliance with the expected values;
- Observing the debug port when a jump is executed (*dbgm_branch*); this mechanism is the one described in this paper, using debug features like the ones available in the ARM processor.

Table 4.2: Experimental results on the MIPS-like processor

	ATE	mem	dbgm_branch	mem+dbgm_branch
Detected	240,591	218,787	48,083	240,547
Fault Coverage(%)	90.30	82.12	18.05	90.29

It is interesting to note that the *dbgm_branch* mechanism proposed in this paper, when combined with the *mem* one, allows a significant increase in the achievable Fault Coverage. In fact, a significant percentage of the faults detected by *dbgm_branch* is not detected by *mem*. Hence, the total number of faults combining the two mechanisms (reported in the rightest column) is higher than both of them, and basically equal to what cab be observed with the first mechanism, which is not usable in in-field test. This result can be explained by recalling that the *dbgm_branch* mechanism allows accessing information about the internal behavior of the processor.

It is finally worth noting that a non-negligible percentage of the faults in the processor (2,429 according to TetraMAX, at least 3,291 according to [64]) are untestable, and thus never produce any misbehavior.

In this work, the usage of some debug features originally introduced in processors was proposed to support software development in order to increase the observability (and hence the fault and defect coverage) during the functional test of PCBAs, with special emphasis on Power-On Self-Test. In particular, these features allow the on-the-fly access to trace information during the test execution, that can be monitored using a special hardware module that may be mapped on an external FPGA which is supposed to exist on the board.

The feasibility and the cost of the solution was demonstrated and evaluated on a Zynq-7000 system by Xilinx equipped with an ARM core. To assess the increase in Fault Coverage, a MIPS-like processor (with modification to mimic the behavior of debug interface in CoreSight architecture) was used whose gate-level netlist is available, and the fault coverage analysis results showed that the proposed technique can achieve figures similar to those obtained during end-of-production testing of the same processor, when it can be tested using an ATE. In this way, the defect coverage that can be obtained during in-field test of a PCBA is greatly enhanced, thus successfully attacking important issues, such as the reduction of *No Failure Found* (NFF). It is also worth noting that our method supports the at-speed test of the system, thus appearing particularly promising especially when dynamic faults represent an issue.

The monitoring IP in the current version is still simple and just uses a subset of the debug features supported by the considered device. However, the feasibility was shown clear for implementing such a module to extract and process information from the processor in a non-intrusive way. Several extensions are possible, leading to the usage of more information provided by the debug and trace facilities, for example, the information as in Fig. 4.3 when exceptions happen can be extracted to monitor the exception handling procedure (which, for sake of simplified miniMIPS experiments, it is not taken into consideration).

Currently further work is being done towards the extension of the method by using further debug features (e.g., extract detailed information in different types of packages instead of send the bytes directly to MISR) and an improved architecture of the monitoring IP. Moreover, a more detailed evaluation of the defect detection capabilities of the proposed approach, e.g. by considering delay fault models, is undergoing.

Part II

Analysis and Mitigation of Single Event Effects on FPGAs

Chapter 5

Introduction

Field Programmable Gate Arrays (FPGAs) are becoming more and more commonly used in various application fields due to their flexibility and low development cost. Furthermore, with technology scaling, the computing power they can provide keeps increasing while the cost and power consumption remains low. This makes them even attractive in safety- and mission-critical fields such as automotive, avionics and space applications.

The FPGA, by different manufacture process technology, can be divided into SRAM-based FPGA, Flash-based FPGA and so on (Antifuse, EEPROM etc. are not in the scope of this work). SRAM-based FPGA, such as the ones from Xilinx and Altera, provide large amount of on-chip resources including Logic Blocks, Digital Signal Processing (DSP) Units, On-chip Memory etc. Together with high performance, low power consumption and high flexibility via partial reconfiguration, these features make SRAM-based FPGA very popular in the market. While comparing to SRAM-based FPGA, Flash-based FPGA does not require extra memory device to store configuration file, and does not require to re-program after each power-on, due to the non-volatile Flash-based configuration memory. More importantly, the configuration memory inside Flash-based FPGA are almost immune to permanent loss of configuration data. Thus, Flash-based FPGA are gaining more and more interest in space and avionic applications.

Though internal architectures may vary from device to device, the SRAM-based FPGA and Flash-based FPGA share the same general architecture as shown in Fig. 5.1, which includes Logic Blocks and Switch Boxes.

- The Logic Blocks topically contains resources such as *Look Up Table* (LUT), Multiplex and Registers and so on that user can configure according to the logic circuit to be implemented and mapped on the FPGA. An example of Logic Block (from Virtex-5 device of Xilinx) is shown in Fig. 5.2.
- The Switch Boxes usually contains interconnection segments which can be configured as active or inactive as required by the routing of the implemented circuit on FPGA. An example of routed design mapped on FPGA (ProASIC3 from Microsemi) is shown in Fig. 5.3.

With other resources provided, such as Block Memories, PLLs etc., designer can create

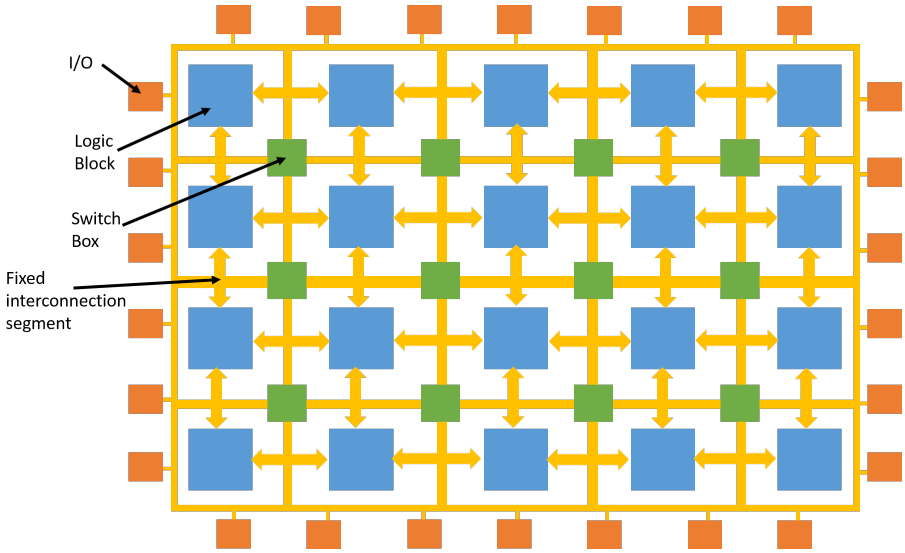


Figure 5.1: FPGA general architecture

complicated system on FPGA using high level *Hardware Description Language* (HDL) or High Level Synthesis methods supported by vendor tools (such as Vivado from Xilinx, Quartus Prime from Altera). A complete design flow usually includes the steps in Fig. 5.4. The synthesis tool compiles the design files in HDL or other high level design format to Gate-Level netlist, and then Place & Route tool is used to map the design to the hardware resources on FPGA. In this process, the netlist could be used for different level simulation to validate the design and timing correctness. Afterwards, the Post-Layout netlist is used by Bitstream Generation tool to generate the bitstream file that can be downloaded to FPGA for implement the design.

5.1 Single Event Effects on FPGAs

With the exponential growth of transistor count, shrinking of transistor size, voltage scaling and increasing operating clock frequency make digital circuits more susceptible to *Single Event Effects* (SEEs). A SEE may occur when an outside disturber, such as radiation particle from space, hits the device causing wrong behavior in the circuit. The energy for a particle to trigger SEE decreases along with the transistor size, while considering SEEs induced by radiation particles at sea level have been reported more than a few years [61], it is mandatory to apply fault tolerant strategies to improve reliability of the system when designing safety- and mission-critical applications.

Two main effects among SEEs considered in this work are *Single Event Upset* (SEU) and *Single Event Transient* (SET). A SEU happens when a charged particle hit a storage element in the design, such as register and memory cell, causing the value stored in the element to change. Meanwhile, A SET happens when a charged particle hit the device (the combinational gate, interconnection resources etc.) causing an abnormal voltage pulse in the circuit that can propagate through the gates till a storage element is reached along the

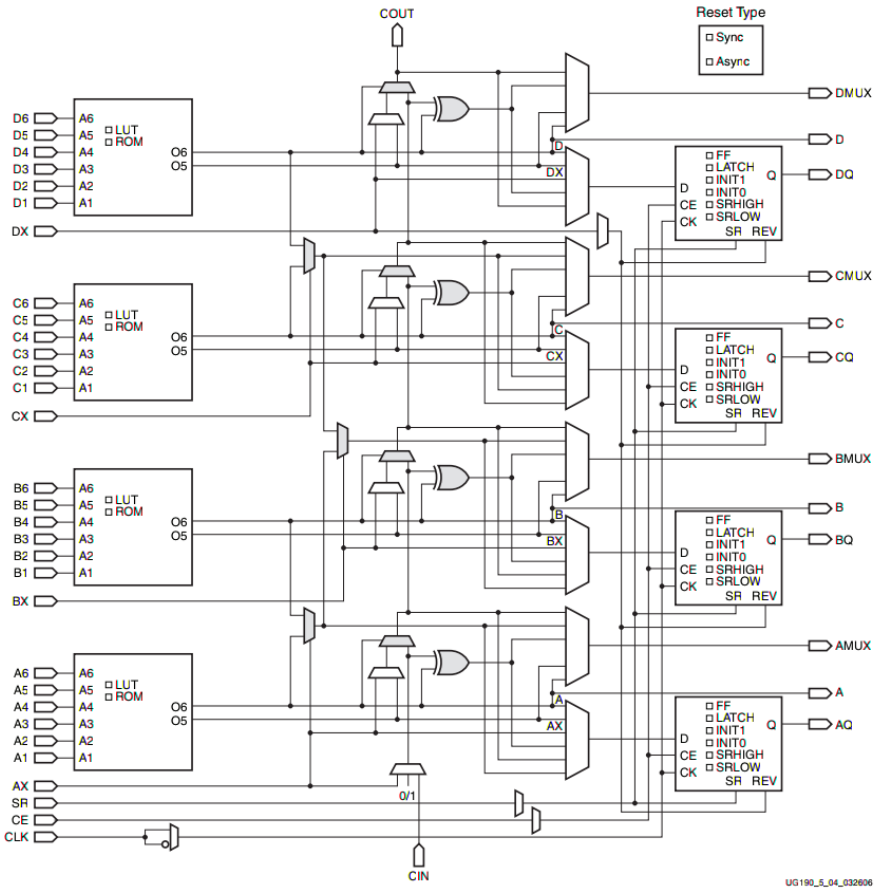


Figure 5.2: Logic Block (SLICEL) diagram from Virtex-5 device of Xilinx [74]

path. And if sampled by the storage elements, then the SET can induce SEU or *Single Event Multiple Upsets* (SEMUs) depending on how many elements are affected.

5.1.1 SEEs on SRAM-based FPGA

As for SRAM-based FPGA, SEU is the main concern due to the vulnerability of its configuration memory. Since the data (i.e. bitstream) in configuration memory contains the configuration data for various hardware resources on the FPGA used in the design, which includes the LUTs, registers and so on in Logic Blocks and programmable interconnection segments in Switch Boxes, a SEU in the configuration memory may cause a permanent error in the mapped design in FPGA depending on whether the bit affected by the SEU is used in the implemented design or not. For an example, taking the Virtex-5 FPGA from Xilinx, a bitflip inside the configuration memory can change a *Programmable Interconnection Point* (PIP) from active to inactive causing the circuit to misbehave as shown in Fig. 5.5.

In SRAM-based FPGA, the configuration memory is made using SRAM cells, while

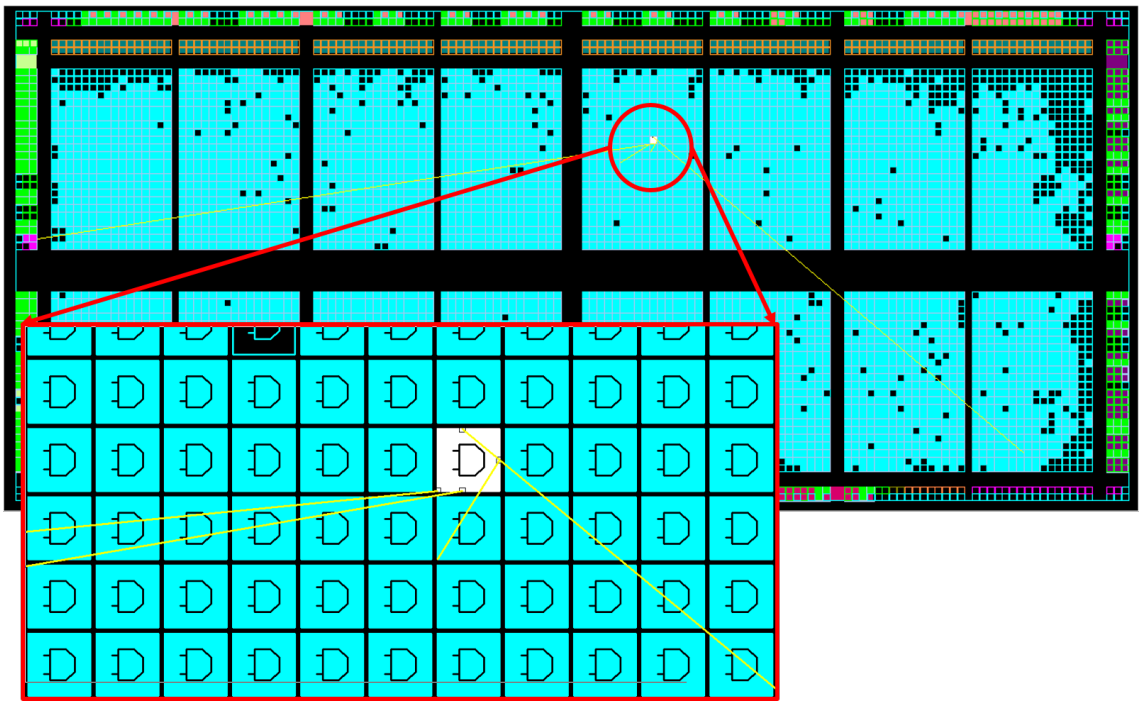


Figure 5.3: A routed design mapped on ProASIC3 from Microsemi

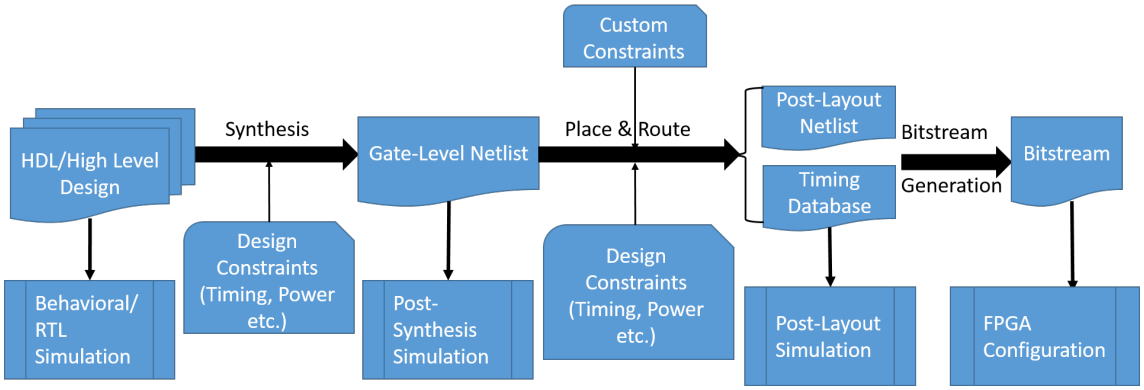


Figure 5.4: Typical FPGA design flow

the SRAM cells are most vulnerable components to SEUs [45], extra caution needs to be applied when SRAM-based FPGA is used.

5.1.2 SEEs on Flash-based FPGA

When it comes to Flash-based FPGA, its configuration memory is made of Flash-based cells which are almost immune to permanent loss of the configuration data. Thus they are becoming increasingly interesting in safety critical fields, in particular for space and avionic applications. However, the floating gate based switches in the Flash-based FPGA

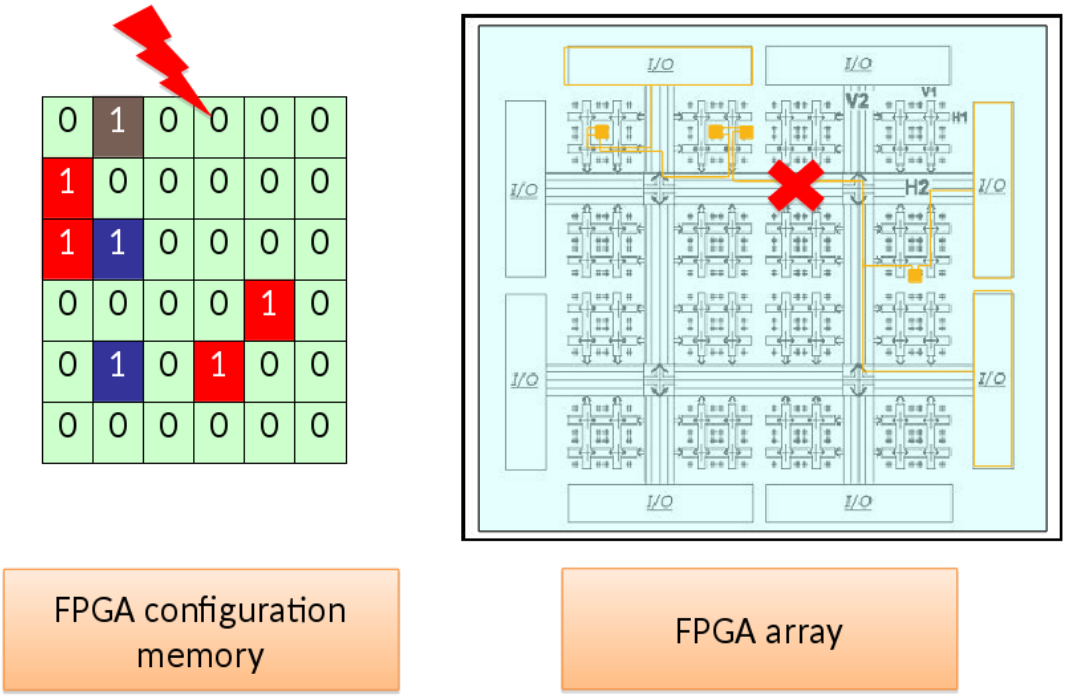


Figure 5.5: A SEU in configuration memory corrupting interconnection

can still suffer transient effects (i.e., SEEs); if hit by high energetic particles, possible critical consequences can be caused on the implemented circuit.

Two distinct effects on Flash-based FPGAs may be identified. The former occurs inside of the floating gate switch: the pass transistor and floating gate transistors usually constitute the floating gate switch. The second occurs when a high charged particle hits a sensitive node of a logic cell belonging to the FPGA's configuration tile. The generated pulse may propagate through the logic depending on the FPGA tile configuration. If the tile is configured to implement a latch, the pulse may turn directly into a SEU because of the feedback paths implemented by the tile logic configuration. Meanwhile if the tile is configured to implement a logic gate, the transient pulse may propagate along the logic paths, reach and be sampled into storage elements corrupting the data stored in them.

Chapter 6

Single Event Effects in SRAM-based FPGA

This chapter discusses the Single Event Effects on SRAM-based FPGA. Due to the SRAM cell's high sensitivity against *Single Event Upset* (SEU) induced by radiation effects, the SRAM-based FPGA is highly susceptible to loss of configuration data in harsh radiation environment, which can lead to critical system failure when used in safety- and mission-critical applications. So the main focus of current chapter is on the analysis and mitigation technique applied to protect the SRAM-based FPGA against SEUs in configuration memory.

The chapter is organized as follows: firstly, a background section is provided with introduction of SEUs in the configuration memory of SRAM-based FPGA, including related approaches for SEU mitigation; then the VERI-Place tool for error rate prediction and SEU mitigation is explained; and finally, two radiation experiments with results analysis is presented to verify the accuracy and effectiveness of proposed analysis and mitigation technique.

6.1 Background

As mentioned in previous chapter, the SRAM cells used for the configuration memory in SRAM-based FPGA are very sensitive to SEUs, which can be induced by a charged particle. And a SEU in the configuration memory can corrupt the functionality of a logic gate (implemented using LUT), a register or an interconnection segment which in turn can cause whole design to misbehave.

There are already several techniques proposed to mitigate the SEUs (and SETs) for SRAM-based FPGA. The most known technique is based on redundancy, such as *Triple Modular Redundancy* (TMR) and its variants. Configuration memory scrubbing is also a well-known technique, in which, the configuration memory is periodically (or triggered by some other mechanism) re-written to correct the SEU, and recently with the high flexibility provided by partial reconfiguration feature, there is technique proposed to detect and correct SEU in the configuration memory in a finer granularity.

6.1.1 Techniques based on redundancy

Among the redundancy-based techniques, the TMR technique involves triplicate the logic in the design and insert a voter before the output as shown in Fig. 6.1. The voter generates the final outputs using a majority vote algorithm $r = xy \vee yz \vee xz$.

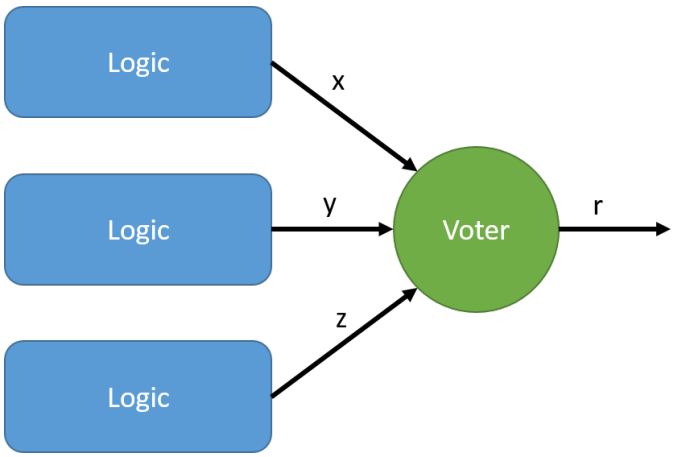


Figure 6.1: General TMR architecture

The TMR could be applied at different level in the system, such as gate level, entity level and system level, where the **Logic** in Fig. 6.1 would be gate (often register or *Flip-Flop* (FF)), system components (e.g., IP) and system itself (e.g., SoC, Chip) respectively.

Different variants of the traditional TMR techniques were proposed over the years to handle different application scenarios. For example, a feedback network can be added to TMR not only to detect which copy of the Logic is misbehaving but to correct it; and in case the error is not possible to fix, the TMR can be downgrade to Duplicate Modular Redundancy mode to detect error and so on.

When TMR is applied at gate level where all the registers are triplicated and voter inserted at the outputs of the registers, SEU can be effectively mitigated, as shown in Fig. 6.2. However, if a SET causes a pulse that can propagate to the registers, all the three registers may be corrupted at the same time and TMR can not provide correct output any more.

With the SRAM-based FPGA from Xilinx, a *Xilinx TMR* (XTMR) tool is provided to automatically apply TMR to the design. And instead of using single majority voter, XTMR triplicates all the design inputs and use three minority voters at the outputs which are also triplicated as shown in Fig. 6.3. And the minority voter's implementation is shown in Fig. 6.4.

Since the XTMR tool triplicates the logic paths in the design directly and only inserts the voters before the final output pins, that means when a SEU occurs in the configuration memory altering one of the paths, the design can still behave correctly. But, if without detect and correct mechanism to fix the corrupted bit in configuration memory, SEU can accumulate overtime, that is, if after some time, another SEU in configuration memory corrupts the other path among the three, the design may fail as illustrated in Fig. 6.5.

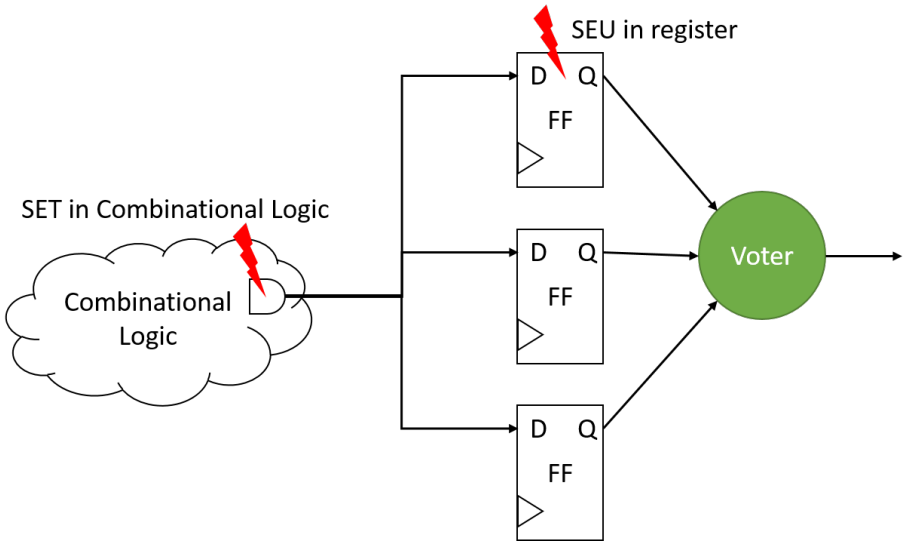


Figure 6.2: TMR applied at gate-level with registers triplicated

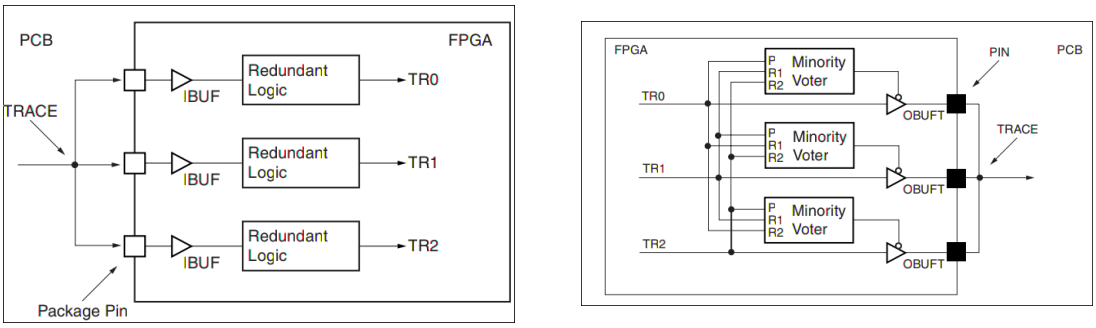
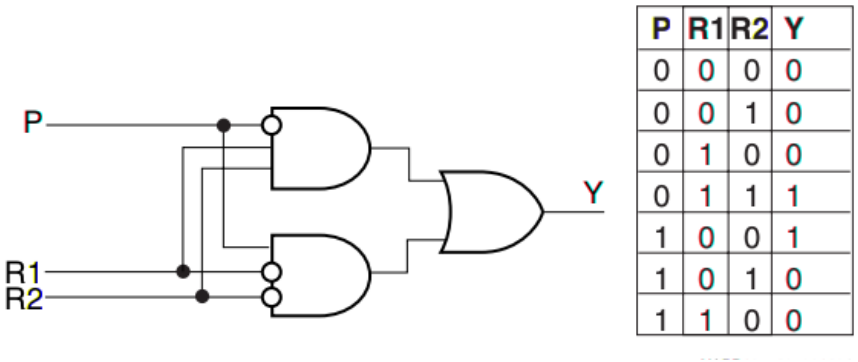


Figure 6.3: Input and output triplication in XTMR [80]



P	R1	R2	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0

Figure 6.4: XTMR minority voter implementation [80]

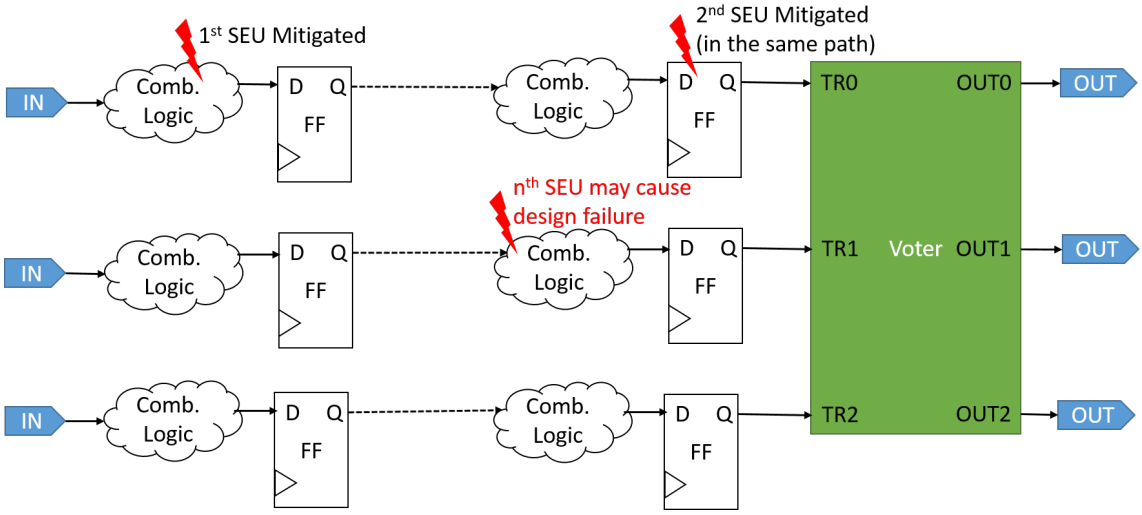


Figure 6.5: Accumulated SEU in configuration memory corrupts design with XTMR

A solution to improve the reliability upon the XTMR applied design is to increase the granularity of the TMR, i.e., divide the logic path into finer segments and insert voters between segments. However, this solution increase the hardware and power consumption overhead significantly due to the extra logic to be introduced into the design. Another solution is to apply scrubbing technique in configuration memory to prevent SEU to be accumulated over time which is discussed in later section.

6.1.2 Configuration memory scrubbing via Partial Reconfiguration

As discussed in previous section, configuration memory scrubbing could be used to correct the SEU in configuration memory and prevent accumulation of SEU. Depending on how the scrubbing is performed, it can be divided into two categories:

1. The entire configuration memory is refreshed by external device. And during the refresh the device can not perform the normal operation and resume after the scrubbing.

2. The configuration memory is refreshed part by part (Partial Reconfiguration). And the design can stay operational (or at least part of it) during this procedure which is enabled the reconfiguration module provided in the FPGA (e.g., *Internal Configuration Access Port* (ICAP) in Xilinx FPGA).

Since the second method provides much higher system availability comparing to the first option, it gained popularity since partial reconfiguration becomes possible in SRAM-based FPGA with granularity at *Frame* level [81] (A *Frame* is a smallest addressable segment in configuration memory, a group of Frames together is used to configure a column of tiles of different type as in Fig. 6.6). However, the mechanism used to trigger the scrubbing and to choose which frames to be refreshed has impact on the performance and *Mean Time To Failure* (MTTF).

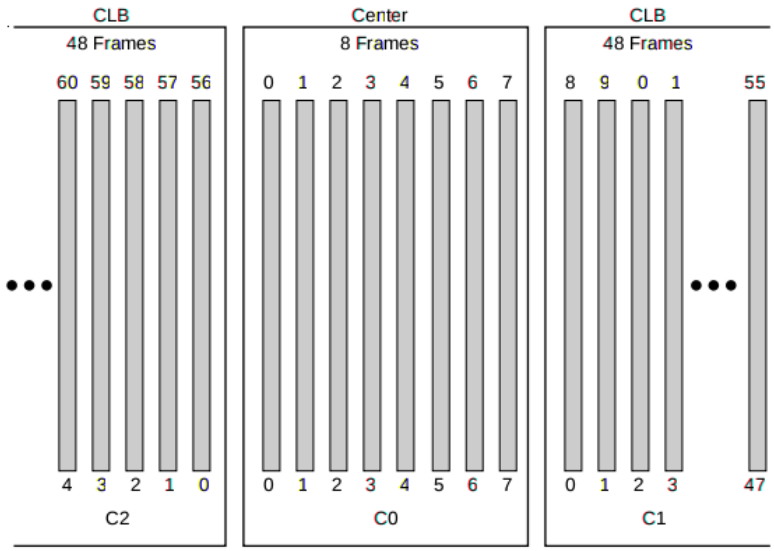


Figure 6.6: Frames in Virtex-5 SRAM-based FPGA's configuration memory [81]

The simplest strategy is to blindly refresh the configuration memory frame by frame [81, 31]. The fault-free configuration data (i.e., bitstream), namely *Golden Copy*, is stored in an external memory device and is read and written to the configuration memory of the FPGA through the ICAP module.

And to detect SEUs in the configuration memory, a *readback* can be performed to read the configuration data from FPGA frame-by-frame to compare with the *Golden Copy*. Then according to the existence of SEU, a frame is decided to be refreshed or not.

Other solutions for scrubbing strategy has been proposed to increase performance and system reliability regarding MTTF, such as in [66, 48]. Thus the analysis regarding the sensitive bits in the configuration memory is valuable for determining the strategy to apply scrubbing.

6.2 Verification and Error Rate Integrated Tool

The *Verification and Error Rate Integrated* (VERI-Place) tool is a software able to perform the Soft-Error Analysis and Placement oriented to the Soft-Error mitigation of circuits on SRAM-based FPGAs[1].

[1]The VERI-Place tool is developed in our research group in Politecnico di Torino by Prof. Luca Sterpone, and currently available online.

6.2.1 Sensitivity analysis with SEUs in configuration memory

The VERI-Place takes the *Xilinx Design Language* (XDL) file, which is native Xilinx netlist format for describing and representing FPGA designs, as input, performs SEE sensitivity analysis and generates several outputs including:

1. Exposure reports, which contain the logic and routing resources related to each *Configurable Logic Block* (CLB). Each resource is considered for being exposed to possible radiation effects, therefore this output reports the overall exposure of the implemented design.

2. Heatmap for sensitive area, where sensitivity is measured by the probability of a failure inside of the specific location, an example is shown in Fig. 6.7 (a and c).

3. Cross-Domain Failure report, which contains the logic and routing resources exposure that may provoke a Cross-Domain Failure affecting the XTMR/TMR design structure (this is only available for design with XTMR/TMR applied).

4. Heatmap for graphical visualization of the interconnection congestion level, which is relative to the whole percentage of PIP and computed for each design, an example is shown in Fig. 6.7 (b and d).

5. Sensitive bit report, which contains the list of sensitive bits with the location of the resources corresponding to the bits. Note this report does not provide the expected Error Rate of the target design.

Furthermore, the VERI-Place tool executes automatically the Error Rate computation by means of a progressive Monte Carlo analysis. The tool analyzes the effects of SEUs within the configuration memory ox Xilinx SRAM-based FPGA, by performing 60,000 random iterations of a number of SEUs ranging from 1 to 500 SEUs. For each number of SEUs, the expected Error Rates for both the original circuit design (*Plain version*) and the version with XTMR applied are calculated, and shown by the tool as "Plain Error Rate" and "Cross-Domain Error Rate" respectively.

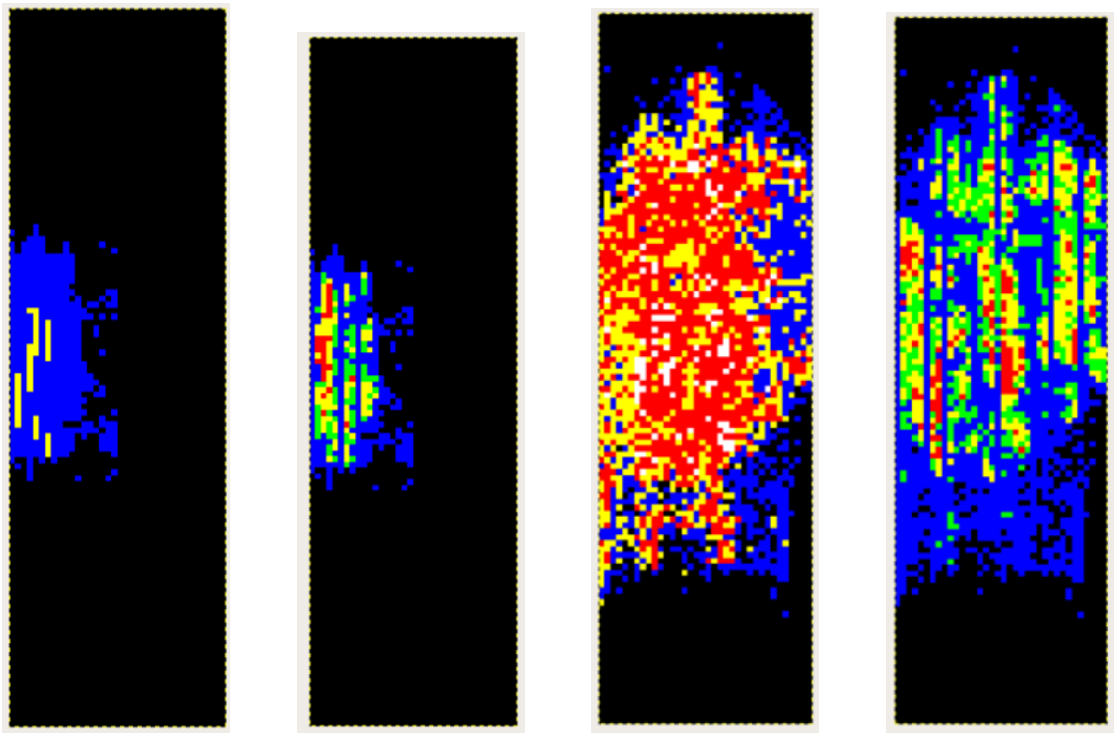
The Error Rate is the ratio between the number of erroneous iterations and the total number of iterations. An erroneous iteration is identified once the analyzed configuration memory bits report an architectural modification that affects the circuit behavior provoking an error on the circuit outputs (independently from the circuit workload).

6.2.2 SEU mitigation with re-placement

The VERI-Place tool is able to execute the replacement and repacking of the circuit in order to be compliant with the Single Fault Assumption full mitigation and to optimize the Maximal Error Gain with respect to the traditional XTMR tool.

Two *User Constraint File* (UCF) are to be generated during this phase:

1. **TMR_area_group.ucf** contains the TMR area division performed on the basis of the TMR single fault assumption.



(a) Heatmap for sensitive area

(b) Heatmap for interconnection congestion level

(c) Heatmap for sensitive area, B14 with XTMR

(d) Heatmap for interconnection congestion level, B14 with XTMR

Figure 6.7: Heatmaps generated by VERI-Place tool on B14 from ITC99 benchmarks [18]

2. **TMR_unpacking.ucf** contains the low level packing of the CLB resource that can be placed on the FPGA using the Xilinx PlanAhead tool.

The two output files can be easily integrated into the Xilinx commercial tool's workflow to generate a new improved implementation of the target design.

6.3 Experiment Analysis

To investigate the accuracy and effectiveness of the analysis and mitigation performed by the VERI-Place tool, two radiation experiments were carried out using the Virtex-5 SRAM-based FPGA from Xilinx.

6.3.1 Radiation experiments with ARM-based SoC on SRAM-based FPGA

One of the experiments[2] is with a simple ARM-based System-on-Chip (SoC) with a Cortex-M0 processor provided by ARM as flattened synthesized netlist through the ARM University Program.

Along with other peripherals such as Digital Clock Management (DCM) and memory peripherals, a UART peripheral was added as a mean of IO device for monitoring the results of the software running on the processor during the experiment. The architecture of the original design is illustrated in Fig. 6.8.

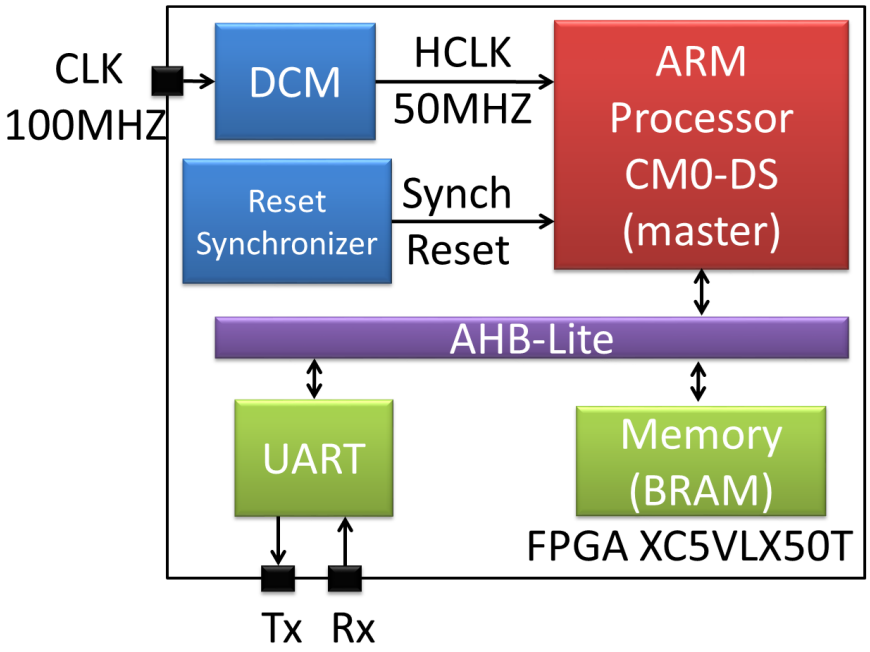


Figure 6.8: Architecture of ARM SoC on Virtex-5 FPGA

Including the original version, totally three versions of the design were prepared and tested, the XTMR version's architecture is shown in Fig. 6.9.

1. *Plain version* includes the original ARM-based SoC design,
2. *XTMR version* is based on the Plain version with XTMR tool applied, by which all the resources in the design (including IOs) are triplicated and voters are inserted to protect the design against SEUs,
3. *XTMR-VP version* is based on the XTMR version with the mitigation from VERI-Place tool applied. The UCF files from the VERI-Place tool are used to generate

[2]The radiation experiments were done in collaboration with European Space Agency

the this version of design, to improve the reliability of the circuit acting only on the Place and Route.

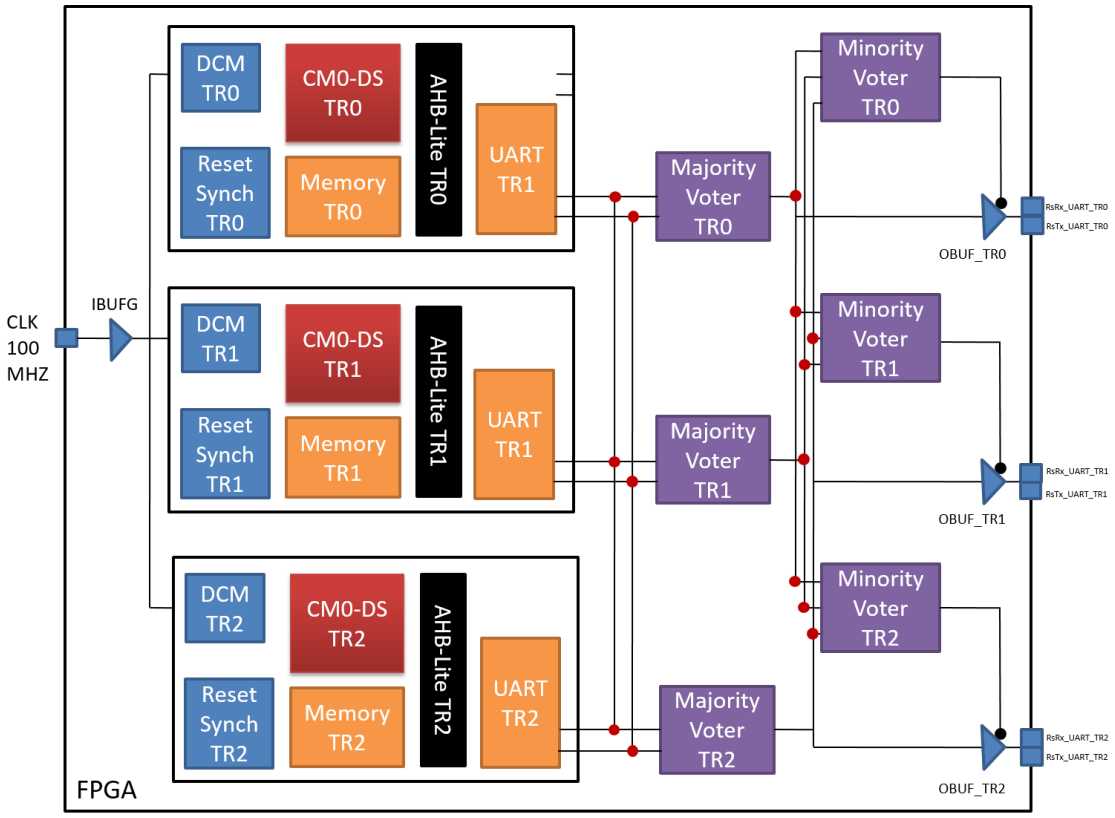


Figure 6.9: Architecture of ARM SoC with XTMR applied on Virtex-5 FPGA

A bubble sort program was used as software benchmark running on the processor (continuously), and results were sent out through UART (in case of the two versions with XTMR, UARTs) to be monitored by a host PC application. The characteristics, in terms of resources, for the three versions of design are reported in Table 6.1, while the physical layout of each version of the design is shown in Fig. 6.10.

Table 6.1: Design characteristics for three versions of ARM SoC

Design Version	LUTs[#]	FFs[#]	BRAM[#]
Plain	3563(12%)	961(3%)	4(6%)
XTMR	13,229(45%)	2887(10%)	12(20%)
XTMR-VP	13,229(45%)	2887(10%)	12(20%)

The radiation experiment was carried out in the Paul Scherrer Institute (PSI), Switzerland, in collaboration with European Space Agency (ESA). The Xilinx Virtex-5 FPGA board was placed under the proton beam with a diameter of 2.5cm , covering only the

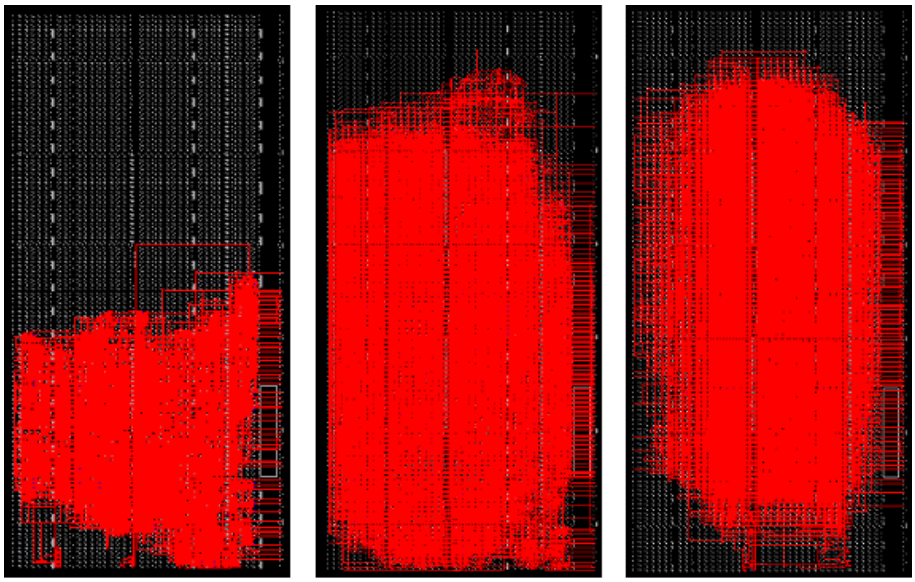


Figure 6.10: Physical layouts showing interconnection networks of a)Plain b)XTMR c) XTMR-VP version of the ARM-SoC from FPGA Editor tool

FPGA chip itself. The flux of proton beam used was about $7.22 \cdot 10^6 p/(cm^2 \cdot s)$ with an energy level of 9MeV.

Since the beam time if expensive in terms of both money and energy, an application was developed to help automating the experiment flow. The application runs on host PC is able to monitor the output data from the UART peripheral (three at the same time in case of XTMR applied) and compare it with the fault-free data gathered before the experiment. In case there is a mismatch, the application will download the configuration memory of the FPGA device by means of bitstream readback operation, record the number of SEUs in the configuration memory and re-flash the FPGA to start a new run of the experiment. The workflow of the host PC application is shown in Fig. 6.11.

Due to the existence of BRAM component in the design (for holding the software code and data), the bitstream readback operation (through Xilinx's *verify* command in iMPACT tool) will corrupt the data in BRAM if the BRAM is used by the ARM processor at the same time. To avoid data corruption, which may lead false error detection through the output, input of the UART component was used to send a command from host PC to put the processor into sleep mode in order to release the control of BRAM before bitstream readback; then another command was sent to wake up the processor to resume normal execution of the program.

As mentioned in previous section, VERI-Place could be used to generate error rate prediction of the design ahead of the radiation experiment. With the data gathered during the radiation experiment and the error rate estimation from VERI-Place tool, if consistency is to be found int he comparison of these two (with same trend), the radiation experiment can be stopped with current configuration and start with a new one (with different design

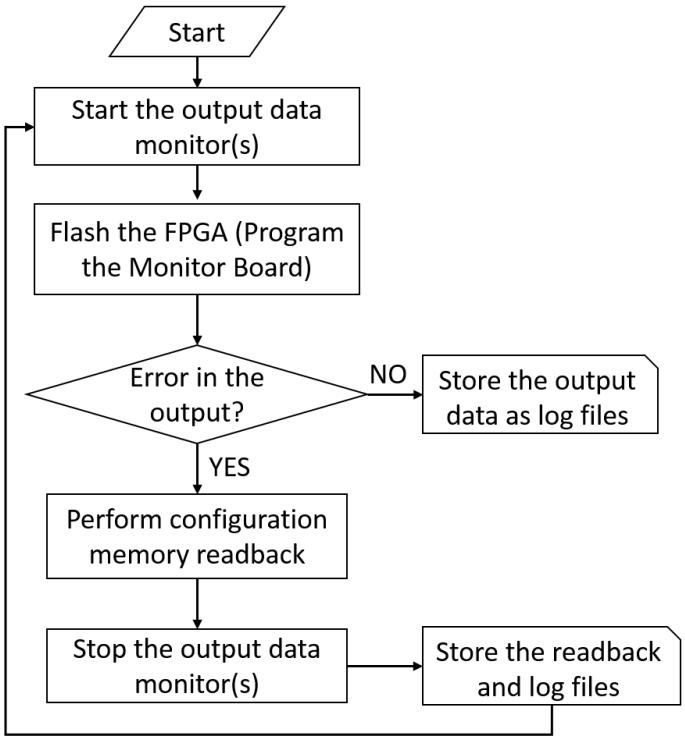


Figure 6.11: Workflow of the host PC application

version, different FPGA board etc.). In this way, the beam time required is reduced, and tests over different configurations of the design can be carried out efficiently, in this case, three versions of the design with two FPGA boards.

6.3.2 Radiation experiment with custom benchmark on SRAM-based FPGA

Another radiation experiment was carried out on the same Virtex-5 FPGA, but with a customized benchmark B13 from ITC99 benchmarks [18].

The B13 circuit is a sequential circuit without memory blocks implementing the interface of weather sensor. Since the B13 circuit itself is quite small comparing to the amount of resources available on the Virtex-5 FPGA, 30 instances of B13 were used together as target design (noted as B13x30).

Also three versions for the B13x30 design were implemented including the Plain version, XTMR version and XTMR-VP version, with the design characteristics reported in Table 6.2. The physical layout of three versions of B13x30 is shown in Fig. 6.12. Different with the setup used for the experiment in PSI, another FPGA board was used to provide input signals to the Virtex-5 FPGA board under test and monitor the outputs.

The radiation experiment was carried out Los Almos Neutron Science Center (LANSCE), also in collaboration with ESA. The SRAM-based Virtex-5 board was placed under the neutron beam, with the flux of neutron set to $5.58 \cdot 10^5 p/(cm^2 \cdot s)$ and energy level

Table 6.2: Design characteristics for three versions of B13x30

Design Version	LUTs[#]	FFs[#]
Plain	1830(6%)	1590(5%)
XTMR	10,841(37%)	4770(16%)
XTMR-VP	10,841(37%)	4770(16%)

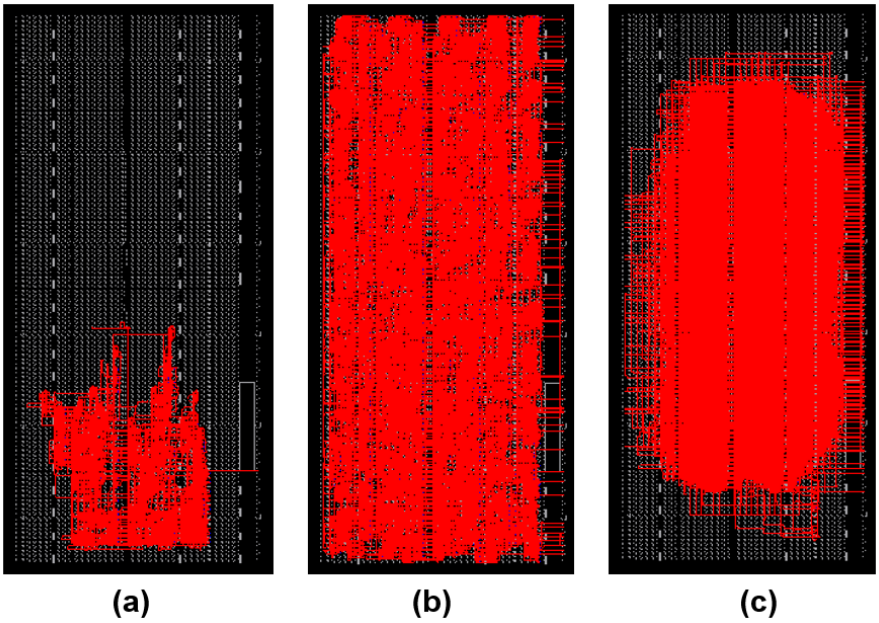


Figure 6.12: Physical layouts showing interconnection network of a)Plain b)XTMR c) XTMR-VP version of the B13x30 from FPGA Editor tool

set to be above 10MeV (this experiment was carried out in parallel with other research groups, with all the boards placed in a row with *Device Under Test* (DUT) aligned with the neutron beam).

A similar host PC application was used to automate the experiment but instead directly communicating with the Virtex-5 FPGA board, the application communicates with the monitor board via UART, which in turn provides the input stimuli to the DUT and monitor the output. In case of an error is detected by the monitor board, the host PC application is notified and a readback of the configuration memory of the Virtex-5 FPGA board is performed to count SEUs in the configuration memory and a re-flash is executed to start a new round of test. Furthermore, a periodical readback was used during experiment to monitor the SEUs accumulated in the configuration memory over time.

6.3.3 Experimental results and analysis

The two radiation experiments allow the collection of several data, including the log files containing the output from the DUT, the readback configuration memory files for counting

the number of SEUs accumulated. The Fluence of the radiation beam was computed and the *Silence Data Corruption* (SDC) cross-section was used as the metric for design reliability assessment, as reported in Table 6.3.

Table 6.3: Fluence and SDC cross-section

Benchmark	Design Version	Fluence	SDC cross-section
ARM SoC (Proton)	Plain	$9.44e10$	$(3.65 \pm 0.01)e-9$
	XTMR	$1.47e11$	$(1.88 \pm 0.01)e-9$
	XTMR-VP	$1.89e11$	$(1.68 \pm 0.01)e-7$
B13x30 (Neutron)	Plain	$1.11e10$	$(1.61 \pm 0.01)e-8$
	XTMR	$1.60e10$	$(1.33 \pm 0.01)e-8$
	XTMR-VP	$2.61e10$	$(1.42 \pm 0.01)e-6$

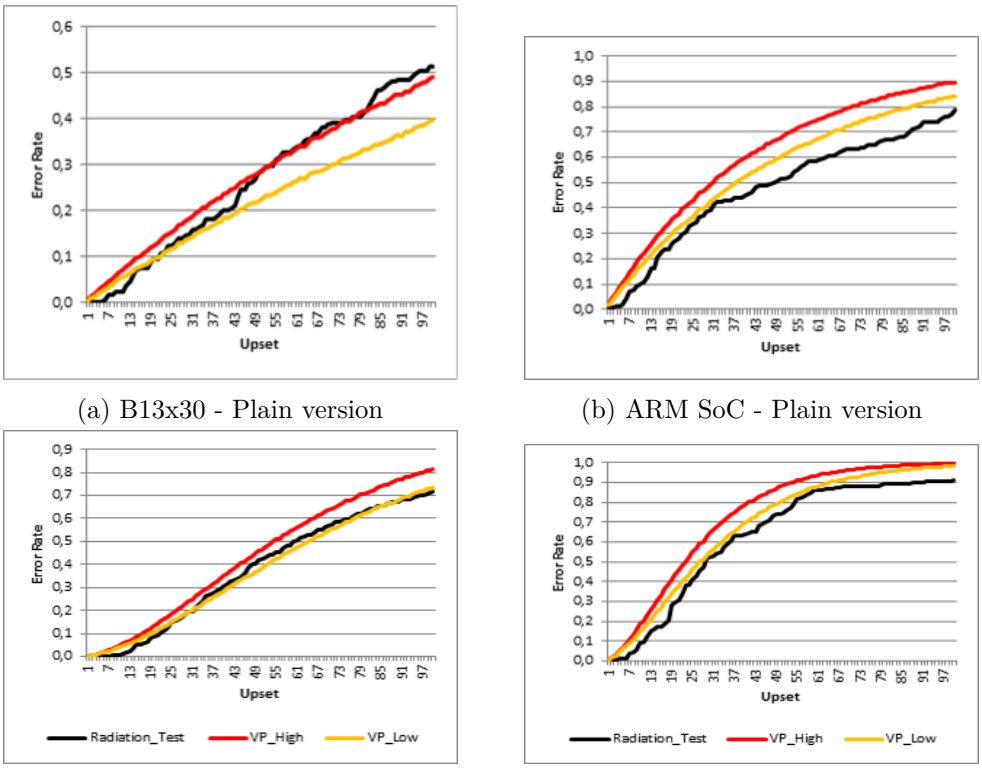
Then the error rate was calculated as the probability of wrong data generated when certain number of SEUs accumulated in the configuration memory. Also the VERI-Place is able to generate the error rate estimation before the radiation test, in which two error rate figures for each design show the error rate predictions under high and low circuit switching activity respectively. The error rates from the radiation test and VERI-Place prediction were plotted in Fig. 6.13[3], where X-axis is the number of SEUs accumulated in the configuration memory while the Y-axis is the corresponding error rate (i.e. the probability to have error in the output).

Please note in the Fig. 6.13 that, the error rate curve with data from radiation experiments fits between the two error rate curves (red curve presents the error rate prediction with high circuit switching activity, yellow with low switching activity) predicted by the VERI-Place tool, however, with an offset. The reason for the offset is that during the radiation experiments, it requires around 3 seconds to re-flash the FPGA device during which time there is no the precise control over the radiation beam so more SEUs accumulated in the configuration memory were counted.

The comparison over the three design versions is shown in Fig. 6.14. It is clear that the version XTMR-VP with VERI-Place tool applied based on the version with XTMR, is the most robust one when SEUs accumulated in the configuration memory (i.e., with the same number of SEUs accumulated in the configuration memory, the XTMR-VP has the lowest error rate).

Meanwhile, when certain number of SEUs accumulated in the configuration memory, the XTMR version actually has worse behavior than the Plain version, due to the granularity the TMR technique is applied by the XTMR tool explained in previous section, and also the XTMR version utilized much more hardware resources in the FPGA leading higher chance the design to be affected by the radiation particles. And this could be tuned by either applying finer granularity with TMR technique or applying configuration memory scrubbing to avoid SEU accumulation.

[3]Currently due to the VERI-Place tool tends to make very pessimistic estimation with XTMR-VP version, so the comparison is not shown here



(a) B13x30 - Plain version

(b) ARM SoC - Plain version

(c) B13x30 - XTMR version

(d) ARM SoC - XTMR version

Figure 6.13: The error rate from radiation experiments and VERI-Place tool

However, when a very few SEUs accumulated in the configuration memory, the XTMR version has the lower error rate than the Plain version as shown in Fig. 6.15 as when only one logic path is corrupted a few SEUs, the XTMR version is still able to deliver correct operations. And number of SEUs corresponding to the point when the error rate of Plain and XTMR design version cross each other, namely breakeven point, together with the radiation profile of the environment the system will be deployed, could be used as a reference metric for tuning the period for configuration memory scrubbing technique.

Two radiation experiments were carried out in two different facilities, with two different radiation profiles, with the same Virtex-5 SRAM-based FPGA from Xilinx. An ARM-based SoC with bubble sort as software application and a customized B13x30 from ITC99 were used as benchmark circuits. Three design versions for each benchmark circuit were used. With the data collected from the experiments, the error rate prediction from VERI-Place tool with the error rate derived from the real data from radiation test were compared. The consistency found in these data validates the accuracy of the error rate prediction from the VERI-Place tool.

In turn, the prediction from the VERI-Place could be used for early stage design reliability assessment to help designer understand the weakpoint of the design against Single Event Effects when SRAM-based FPGA is to be used in harsh radiation environment.

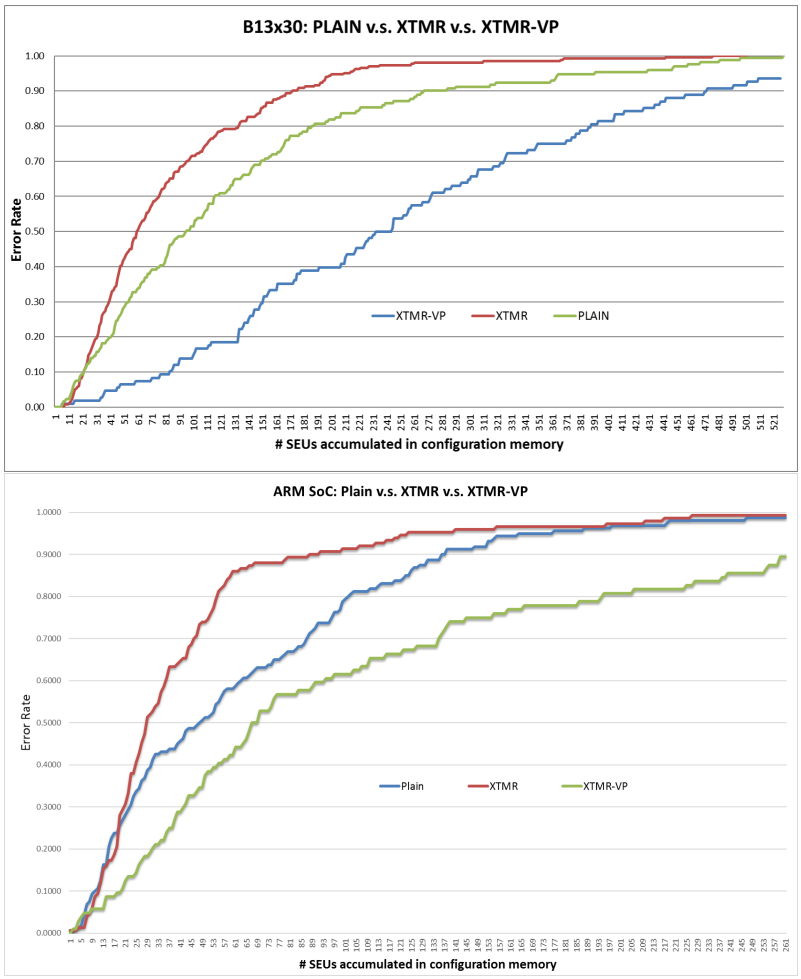


Figure 6.14: The SDC error rate comparison over three design versions

And the error rate prediction could also be used as reference metric for other fault tolerant techniques (such as configuration memory scrubbing).

Furthermore, the VERI-Place is able to generate constraint files for Place & Route, which can be easily integrated into commercial tool, for generating an improved version of the target design, which is verified with the comparison across the three design versions during the radiation experiments.

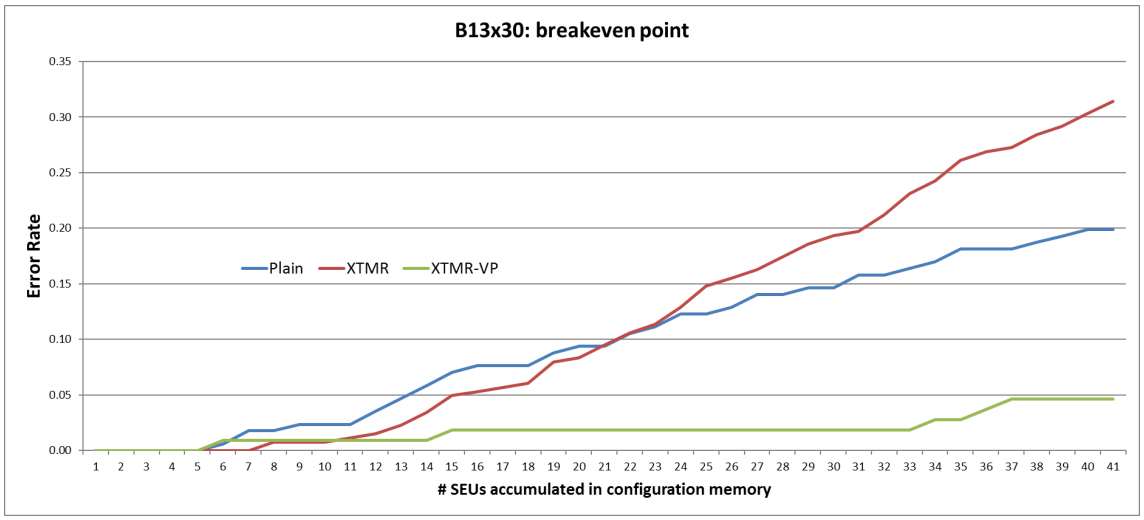


Figure 6.15: Breakeven point in case of B13x30 benchmark

Chapter 7

Single Event Effects on Flash-based FPGA

This chapter discusses SEEs analysis and mitigation techniques on Flash-based FPGA[1]. Since the configuration memory in Flash-based FPGA is almost immune to SEUs, the main focus of this chapter is on the *Single Event Transients* (SETs). The chapter is organized as follows: firstly, a background regarding Flash-based FPGA and the radiation induced Single Event Effects is given, including previously proposed approaches; then the proposed SEE analysis and mitigation flow including a SET analytical model, a FPGA logic and route model, a SET Analyzer and a SET-aware place and route tool is described in separate sections; and finally, the analysis experiment and radiation experiment to verify the proposed flow is explained.

7.1 Background

Due to the non-volatile configuration memory, the Flash-based FPGA are commonly used space and avionic applications. However, these devices are composed of floating gate based switches that can suffer SETs, if hit by high energetic particles, provoking possible critical consequences. An SET may cause the circuit mapped on the FPGA to misbehave if it is able to propagate through the logic paths and eventually be sampled by memory element corrupting the value previously stored inside (or directly corrupt the output signal).

In the last years, science researchers have acknowledged SET as a forthcoming issue in digital technologies, especially due to the technology shrink [63]. Several works investigated the nature of SETs on Flash-based FPGAs, analyzing the propagation of the transient pulse through the combinational logic data path and routing resources [77, 62]. Previous works reported radiation test experiment [14] and electrical fault injection [70] of SET propagating on custom circuits designed specifically to observe SETs, also some

[1]This work has been done with collaboration with ESA and Microsemi; And part of the research tools and results has been used in collaboration with CGS S.p.A within the European Elucid space mission project

results on SET dependency on clock frequency have been presented in [11]. Recent experiments of accurate SET pulse electrical injection [71] show a strong SET pulse-width modulation when SET pulses traverse logic gates. Besides, it has been observed that the SET pulse width at the input of a storage element is strictly dependent on the propagation and type of traversed logic gates [63].

7.1.1 SET pulse profile in Flash-based FPGA

A SET appears when an amount of current causing a voltage glitch of elevated magnitude is generated. Generally, this happens when a charged particle crosses a junction area. The voltage glitch propagates for notable distances and becomes indistinguishable from normal signal traversing combinational gates and routing interconnections. As far as sub-nanometer technology is considered, the main source of SETs corresponds to the combinational logic, since the effects are generated by the reversed biased junction collection charge accumulated in the sensitive area of logic gates.

Regarding Flash-based FPGAs, two distinct effects may be identified. The former occurs inside of the floating gate switch: the pass transistor and floating gate transistors usually constitute the floating gate switch. The second occurs when a high charged particle hits a sensitive node of a logic cell belonging to the FPGA's configuration tile. The generated pulse may propagate through the logic depending on the FPGA tile configuration. If the tile is configured to implement a latch, the pulse may turn directly into a SEU because of the feedback paths implemented by the tile logic configuration. Meanwhile if the tile is configured to implement a logic gate, the transient pulse is assumed to be propagated only if the voltage glitch generated by the particle hit on the struck node changes by more than $V_{DD}/2$ [79]. Once a SET is generated into the sensitive area of a logic gate it starts its propagation through the logic paths until a sequential element is reached. During its propagation the SET pulse may pass through inverting (i.e. INV, NAND, NOR ...) and non-inverting (i.e. AND, OR ...) gates. The SET propagation through logic gates undergo to different electrical phenomena that affect the shape of the pulse modifying its voltage amplitude, the width and the speed along the traversed logic path [11, 71].

SET propagation in Flash-based FPGAs has been investigated by electrical injection and radiation tests [70], where the analysis denoted that while traversing inverting gates, both the amplitude and the duration of the SET are broadened or filtered depending on the number and type of crossed-gates. Recently a full characterization of the Microsemi Versatile logic gates has been performed by developing an analytical model for the electrical simulation considering the Propagation Induced Pulse Broadening (PIPB) effect. Considering the shape of an SET effect, which an example of its propagation through an inverting gate is illustrated in Fig. 7.1, given an SET transition 0-1-0, the developed analytical model is able to dynamically describe a broadening coefficient $C(x) = \Delta t^U(x) - \Delta t^I(x)$, where in case $C(x) > 0$ the SET pulse is broadened, otherwise attenuated.

Although the dynamic model of the PIPB effects has been proven to be an effective solution to analyze the SET propagation in Flash-based FPGAs, it is not efficient nor suitable when it comes to full analysis of the SET sensitivity of complex circuits having several thousands of logic gates and FFs fully interconnected in millions of logic paths,

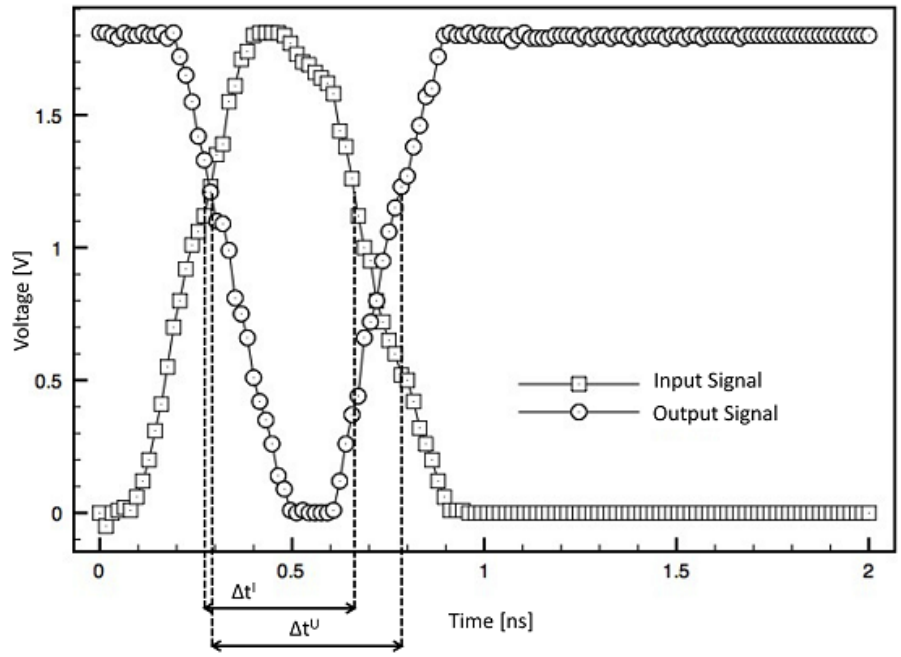


Figure 7.1: The SET propagation through an inverting gate with an input transition 0-1-0

since a complete SET analysis with this dynamic model would require an extremely huge simulation time which will be impractical. In the present work, an algorithm was developed that allows to perform automatic SET sensitivity analysis of complex circuits mapped on Flash-based FPGAs in an optimal time and is able to provide accurate results.

7.1.2 Previous analysis and mitigation techniques for SEEs on Flash-based FPGA

Several radiation test campaigns had proven the immunity of commercial Flash-based FPGAs to upsets in their configuration memory cell [62]. Several works addressed the problem of SEUs in the user memory resources. Error Correcting Code (ECC) [78] is a conventional solution to protect soft errors in SRAM, while Triple Modular Redundancy (TMR) is the most widely applied technique for mitigating SEUs in logic memory. On the other hand, SET effects in CMOS Integrated Circuits (ICs) have become a severe concern in all Deep SubMicron (DSM) technologies. The principal technology factors that make ICs more sensitive to transient pulses generated by energetic particles are the smaller dimension of transistor size and the reduced thickness of interconnections. Indeed, the progressive technology shrinking induces the simultaneous reduction of both the circuit node capacitance and voltage levels.

When Flash-based FPGAs are considered, the number of sensitive nodes belonging to a single configurable cell (also referred as *Versatile*) is drastically greater than standard ASICs since a cell has several configuration points that have been shown to be critically sensitive. Several works have been proposed in the past in order to analyze and to mitigate

the problem of SET. The first kind of methods is based on the classical fault-tolerant approach such as TMR [12]. Other techniques have been proposed relying on replication design methodology by using time or spatial redundancy. For example, in [44] the FF implementation is divided into two latches block by synthesis tools, through which the original FF is modified including a dual-sampling latch with delayed signal sampling able to filter the SET effects.

However, the usage of checkers and logic duplication inherently introduces significant delay, area and power overhead. Since the overhead introduced by these techniques is dramatically high due to the full logic replication, less area expensive solutions have been proposed aiming at directly modifying the configuration of the Versatile sensitive nodes by changing their configuration memory pattern [1]. Such kind of techniques firstly analyzes the sensitive nodes of the mapped circuit basing on the probability of having a critical location. All the selected sensitive gates are then reconfigured by changing the configuration memory pattern without changing the Versatile implemented logic function. Nevertheless these techniques provide effective solutions, they all rely on full redundancy since the advantages of logic function patterns modification are very limited.

As reported in [68], place and route algorithms are a viable solution to mitigate SET effects on Flash-based FPGAs since they can be adopted at the application level without requiring modification of the logic cells and FFs configuration, however such solutions must be rightly calibrated in order to optimize their capabilities of reducing the PIPB effect and must be corroborated by suitable redundancy and filtering techniques in order to avoid the impact of both SEUs and SETs.

A preliminary place and route algorithm developed for this specific purpose has been proposed in [68], while a placement re-timing algorithm has been proposed in [82]. However, an accurate radiation experiment evaluation of this approach [69] showed that it is only capable to partially reduce the overall SET sensitivity of a circuit, while most of the transient errors are bypassing the filtering optimizations. The main limitation of these previous approaches is due to the application of the mitigation strategies on an already available place and route solution. Indeed, such kind of techniques firstly analyzes the sensitive nodes of an already mapped circuit basing on the probability of having a sensitive node on specific critical logic gates, secondly the mapper and the place and route algorithms are applied on the previously estimated sensitive nodes in order to minimize the SET sensitivity. A second relevant limitation of the previously developed methods is characterized by the absence of detailed routing data, since both routing infrastructure and routing algorithm are not available. The missing availability of routing information and algorithm provokes an evident limit on the applicability of transient filtering, since routing segment capacitive and resistive loads are only estimated by placement. So a detailed FPGA logic and routing model that allows the execution of both placement and routing algorithms on the basis of the target Flash-based FPGA architecture is needed.

7.2 A complete flow for analysis and mitigation of SETs for Flash-based FPGA

In this work, a complete flow was proposed including the analytical model for the SET propagation along with the logic and routing model, and a SEE-aware place and route algorithm for SEE mitigation for Flash-based FPGA. The flow is illustrate in the Fig. 7.2.

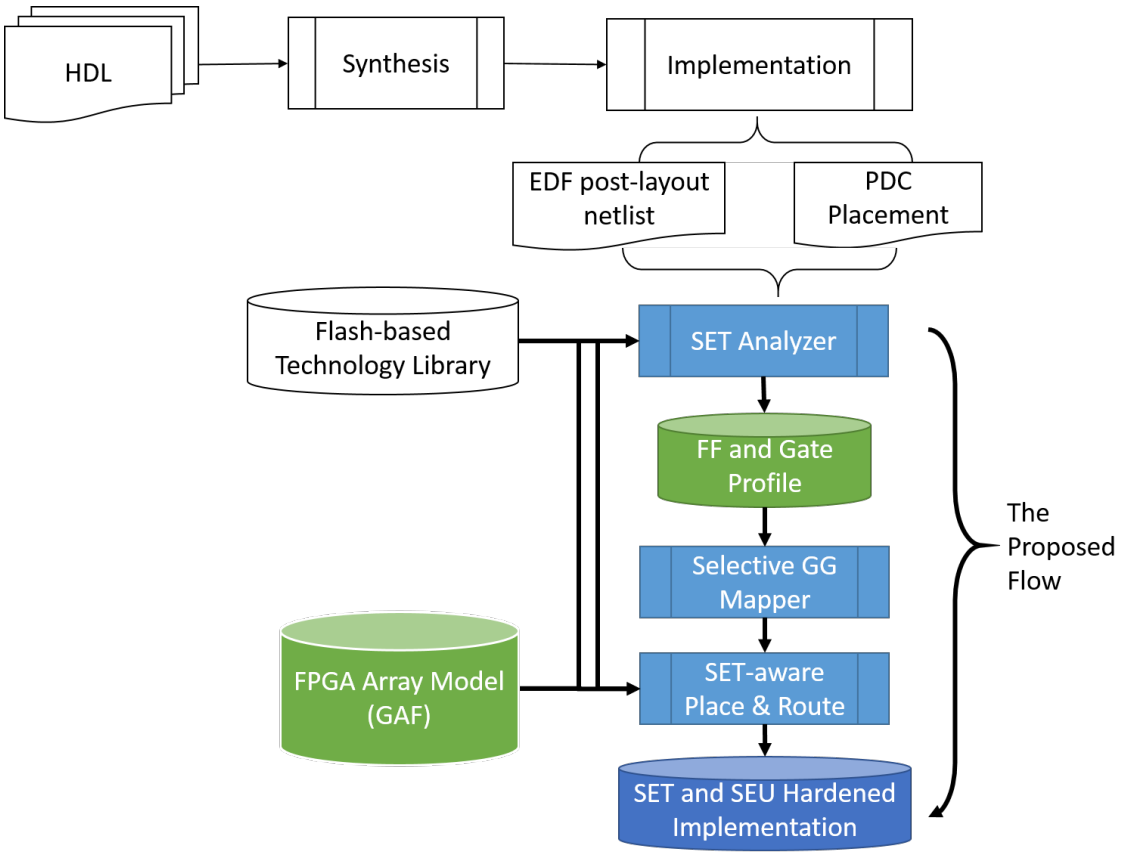


Figure 7.2: The proposed flow for analysis and mitigation of SEEs in Flash-based FPGA

In order to apply the proposed flow, it is required to use a commercial tool chain able to generate a post-layout netlist (*EDF netlist*) and a *Physical Design Constraints* (PDC) placement file (in case of Microsemi Flash-based FPGA). The netlist file contains the full functional description of the circuits at the FPGA physical level, in the form of Versatile logic gates, FFs and interconnection links, while the placement file contains all the locations of the logic resources and input/output pins on the FPGA logic array. These two files are commonly generated by all FPGA manufactures (may in a different file format with the same information). Generally they start from a circuit design in HDL and through synthesis and implementation tools they generate post-layout EDF and PDC files.

Our flow starts with elaborating the EDIF post-layout netlist and the PDC placement

locations of the circuit by means of the SETA tool, which consists of an algorithm able to perform an exhaustive evaluation of SET effects on all the sensitive nodes of a circuit mapped on Flash-based FPGAs. The results of the SETA tool are two profile databases: the Flip-Flops maximal sensitivity list database and the Gate To Gate broadening coefficient database. The former reports for each FF the maximal pulse width observed on its input, while the latter reports the broadening width coefficient between each couple of gates through all the circuit data paths. The maximal pulse width is a number that may vary from 0 to several nanoseconds, while the broadening coefficient may be a positive or a negative number depending if the effect is filtering (negative) or broadening (positive).

Once the profiles are generated, the SET path placer algorithm is executed. The algorithm performs a placement of the logic gates and FFs regulating their locations by a timing and capacitive load metric able to achieve the reduction of PIPB effect between each couple of gates through the circuit data paths.

Then, the *Selective Guard Gate* (SGG) mapper is executed. This mapper is able to modify the circuit post-layout netlist in two ways. The first modification is related on the FFs profile: if the FF is considered sensitive, the algorithm triplicates the FF and inserts a voter structure. The second modification is related to the pulse propagation: if the pulse is considered critical, it inserts a suitable guard gate filtering structure on the inputs of all the FFs that have not been protected by the SET path placer and according to the SET pulse width reported by the FFs database profile. Both the modifications are performed acting on the circuit EDIF netlist file. Finally, according to the results of the mapper and of the placer a new placement constraint and netlist files are generated. The generated files can be used by standard Flash-based FPGA implementation flow.

7.2.1 Analytical SET nanometer model

An accurate modeling of the SET phenomena generated by radiation particles within the silicon structure of nanometer devices was proposed. The method consists of three phases: the generation of the SET pulse phenomena which is modeled as transient pulse shape, the localization of all the combinational gates within the circuit description and finally the execution of the propagation of the SET pulse starting from each sensitive node of the circuit and traversing the logic gates and routing interconnections until an input of a storage element (i.e., a Flip-Flop or a Memory Bank) is reached. The model allows the identification of the expected SET width and allows estimation of the global sensitivity of the circuit. To the best knowledge of the authors, this model is the first solution that is able to integrate physical design analysis and Matlab computations in order to evaluate the dynamic behavior of a VLSI device.

SET generation and analytical model

In order to generate the pulse shape, the developed model elaborates the physical layout description of each circuit logic gate, described by standard Graphic Database System for IC layout (GDS-I), which represents a 3D model of the implemented circuit. The model consists of 3 phases described by Matlab code. The first phase, based on the

characterization that was provided in [70], generates the SET model according to the definition depicted by the shape tn in Fig. 7.3.

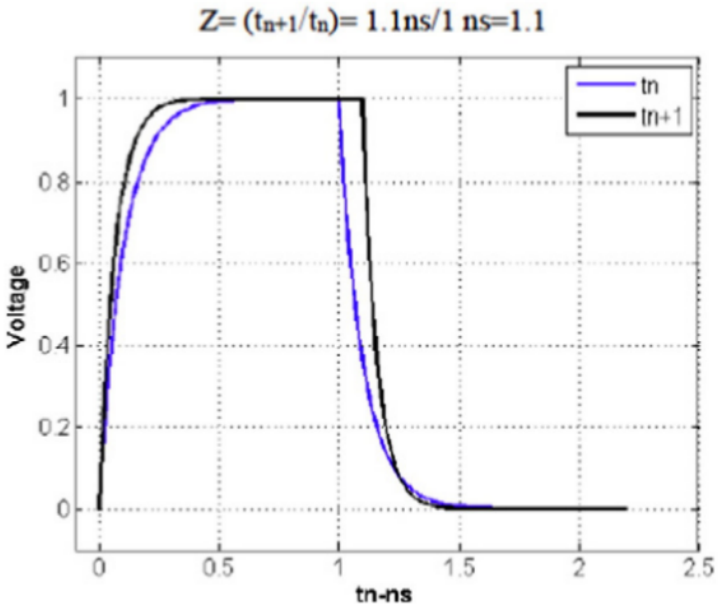


Figure 7.3: SET pulse shape modeling the original pulse (i.e., positive transition) generated from the GDS-I model (t_n) and after the propagation through a logic gate (t_{n+1})

The second phase executes the propagation on the basis of the Resistive and Capacitive load calculated on the GDS-I 3D model of the circuit. The propagation coefficient is used in the model reported in Eq. 7.1 in order to generate the expected propagation coefficients for all the logic paths [71]. Please note that when the coefficient is lower than 0 the signal is filtered, and vice versa the original pulse is broadened. In Fig. 7.3, the pulse shape t_{n+1} is obtained in case of broadening. The third phase includes the execution of the propagation and on the classification of each Flip-Flop sensitivity. The model has been used to reduce the sensitivity of the benchmark circuits that used for the experimental evaluation.

$$T_{n+1} = \begin{cases} 0, & \text{if } (T_n < kt_p) \\ T_n + \Delta t_p, & \text{if } (T_n > (k+3)t_p) \\ \frac{(T_n^2 - T_p^2)}{T_n} + \Delta t_p, & \text{if } ((k+1)t_p < T_n < (k+3)t_p) \\ (k+1)t_p(1 - e^{k - \frac{T_n}{t_p}}) + \Delta t_p, & \text{if } (kt_p < T_n < (k+1)t_p) \end{cases} \tag{7.1}$$

SET characterization

The Single Event Transients (SETs) characterization has been performed using the SETA tool, which has been implemented in [69] which is an algorithm that performs the SET

propagation analysis of a circuit mapped on a Flash-based FPGA. It consists of two phases: the former locates all circuit combinational gates and identifies their propagation nodes until a storage element (a FF or a Latch) is reached, the latter performs the propagation of a SET pulse starting from each sensitive node and traversing all the circuit logic path and storing the maximal length observed by the SET pulse at the input of each storage element. The results are stored in two databases reporting the maximal SET pulse at the input of each FF and the broadening coefficient between the couple of gates.

7.2.2 FPGA logic and routing model

Contemporary FPGA architectures are generally characterized by the well-known island-style FPGA model [16] including a two-dimensional array of logic elements that are interconnected via a programmable routing network. On the basis of these resources organization, FPGA families may have different architecture depending on the number of logic elements, local and global routing wires as well as pin locations and short wires granularity. The present section describes the FPGA logic and routing model developed for supporting the placement and routing algorithm oriented to the SET mitigation. The model has been developed following a parametric format named *Generic Array Format* (GAF) is supported by a two dimensions matrix mesh format where all the FPGA resources refer too. Basically, all the resources are defined by nodes and lines organized on the two dimensions mesh space. The parametric format, whose graphical organization is illustrated in Fig. 7.4, allows the definition of the following FPGA characteristics:

1. *2-D mesh matrix space*: it defines the maximal space area and the number of logic elements and switch matrix column and rows. This parameter allows also the identification of FPGA regions that are not used for reconfigurable switches (e.g., in particular when these areas are used for embedded hardwired microprocessors, memory blocks or DSP modules).

2. *Switch matrix and logic element block*: it defines the number, dimension and organizations of the routing and logic element areas. A traditional island style FPGA has generally one or more logic elements for each switch matrix block, our model allows different organizations where routing segments are divided into more switch blocks and logic elements are organized in different position. Please note that since both the switch matrix block and the logic elements are defined by rectangular shape with proper dimensions on the 2D-mesh, our model supports any kind of FPGA architecture and it can be eventually adapted to novel FPGA logic and routing topology.

3. *Internal routing resources*: it defines the programmable interconnection points of a switch matrix block. Each resource is defined by a segment with a source and destination point that can be placed according to the user definition in any position of the two dimensions mesh matrix (e.g., reasonably source and destination points are placed on the same switch matrix frame).

4. *Hardwired routing resources*: it defines the starting and ending points of all the hardwired lines on the FPGA architecture. It includes the horizontal and vertical

hardwires lines, horizontal and vertical long lines as well as the logic element input/outputs. Each single routing line is defined by a set of points identifying the source point and the destination points. Please note that the position of the source and destination points must correspond to the one of the internal routing resource.

5. *Input/Output pins*: it defines the position of the I/O pins. They can be located on all the available two dimensions mesh matrix space allowing the modeling of all the kinds of FPGA pin-out organizations.

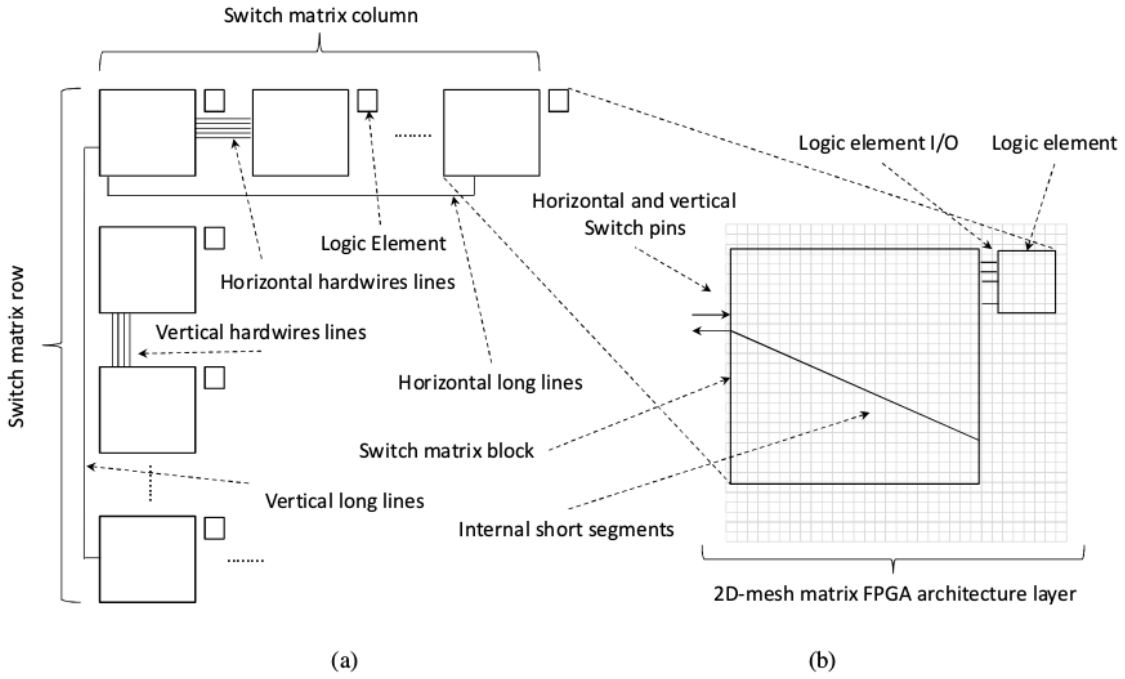


Figure 7.4: Parametric architectural FPGA model for mesh-matrix oriented place and route algorithms (a) and the mesh matrix format in two-dimension (b)

The GAF parametric architectural FPGA model has been defined within a textual file where it is possible to specify regular or not regular resources, by this way it is not necessary to define internal short segments for all the switch matrix elements, since they are replicated on all of them, and, using the same model, it is possible to define singular routing and logic resources. The main advantage of this format is the extremely high flexibility to be adapted to various kinds of FPGA architectures covering almost all the state-of-the-art FPGA devices.

7.2.3 SETA: Single Event Transient Analyzer

The SETA tool firstly loads the circuit netlist description and the PDC file for the physical location of each logic gate and input/output pin. Secondly, it creates a *Physical Design*

Description (PDD) file of the circuit consisting in a direct graph where logic gates are modeled as vertices while interconnections as edges, using a tool developed called AFL2PDD. The *Actel Flattened Netlist* (AFL) netlist format is a traditional flattened netlist including detailed FPGA gate level description and nets between cells that can be generated through the Synopsys synthesizer embedded within the Microsemi Libero SoC tool.

Furthermore, the tool generates a logic and routing global graph integrating the information read from PDC file with information related to the FPGA technology library of the target device. For this purpose, the logic and routing model described above was used with the data previously collected during the characterization performed in [70] and calibrated in the model presented in [11]. The final logic and routing graph consists of a directed graph structure where I/O pins, FFs, RAM or ROM pins are considered as *terminal points*, while combinational logic gates are considered as *crossing points*. Interconnections are defined as weighted direct edges between nodes. The weight of each edge provides information on the resistive and capacitive load of each FPGA interconnection segment.

To analyze SET sensitivity of the circuit, the main SETA algorithm executes in two steps: 1) locates all circuit combinational gates and identify their propagation nodes until a storage element (a FF or a Latch); 2) performs the propagation of a SET pulse starting from each sensitive node and traversing all the circuit logic path and storing the maximal length observed by the SET pulse at a the input of each storage element. The generation and propagation of SETs is performed by the algorithm depicted in Fig. 7.5.

```
// Step 1:Single Event Transient Generation
SET GP = generate_list_SET();
// Step 2:Single Event Transient Propagation
for each pulse P ϵ SET GP {
      for each sensitive node I ϵ SN {
            apply pulse P to I;
            find destination node DN ϵ (SN, I);
            for each destination node DN
                  propagate P on (I, DN)
      }
}
```

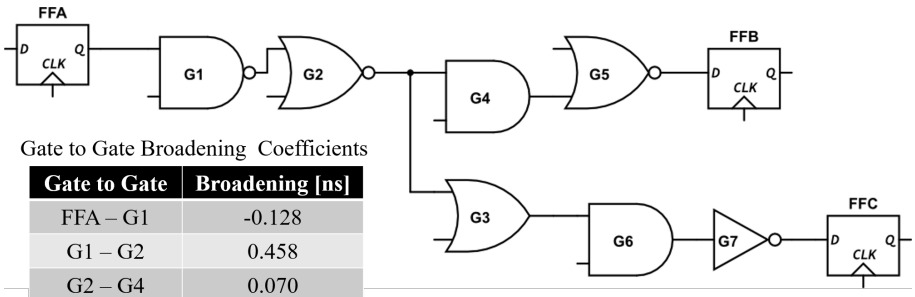
Figure 7.5: The main SETA algorithm steps

The first step is executed by the function *generate_list_SET* that creates a list of original pulse shapes derived from a defined voltage amplitude and a time duration. Each pulse shape is described by 100,000 points, allowing a precision of 1 ps. The second step consists in the propagation of the transient pulse. Each original SET pulse is applied to each circuit sensitive node and propagated.

The propagation function is the core of the developed algorithm. At each recursive

iteration, the pulse, described as a voltage array P, is propagated through the crossing-points (i.e. combinational gates) up to the next terminal point (i.e. a sequential element). When crossing a combinational gate the source input voltage array is transformed into a drain output voltage array. The transformation of the pulse propagation is the key-feature of the proposed algorithm. Taking the analytical SET nanometer model with the several routing parameters integrated into PDD graph with the FPGA logic and route model, it is able to calculate the PIPB coefficient during each SET propagation.

The results of the SET analyzer algorithm are stored in two databases reporting the maximal SET pulse at the input of each FF and the broadening coefficient between all the couples of gates. An example of such results is shown in Fig. 7.6.



Gate to Gate Broadening Coefficients

Gate to Gate	Broadening [ns]
FFA – G1	-0.128
G1 – G2	0.458
G2 – G4	0.070
G2 – G3	-0.090
G3 – G6	0.480
G6 – G7	0.092
G7 – FFC	0.140
G4 – G5	-0.094
G5 – FFB	0.130

FFs Maximal Broadened Pulse Width

Flip-Flop	Maximal Pulse [ns]
FFB	0.436
FFC	0.952

Figure 7.6: Example circuit and results from SETA tool

7.2.4 Selective Guard Gate mapper

After the two databases generated by the SETA tool, a *Selective Guarding Gate* (SGG) mapper is used for SEE mitigation before the SET-aware Place & Route tool is applied.

The SGG mapper performs two modifications against the original design, based on the information in the two databases provided by the SETA tool in previous step:

1. If the SETA tool reports a register (FF) as SEU sensitive, the SGG mapper will insert a TMR logic structure including a majority voter to protect the sequential element against SEU.

2. Depending on the sensitivity estimated by the SETA tool and the expected maximal SET pulse reaching at the register's input, the SGG mapper will insert a combinational *Guard Gate* (GG) logic structure before the input of the selected register, for example, as in Fig. 7.7. This modification is applied to those registers exceeding the expected sensitivity. And the overhead of each GG structure is related to the maximal pulse length.

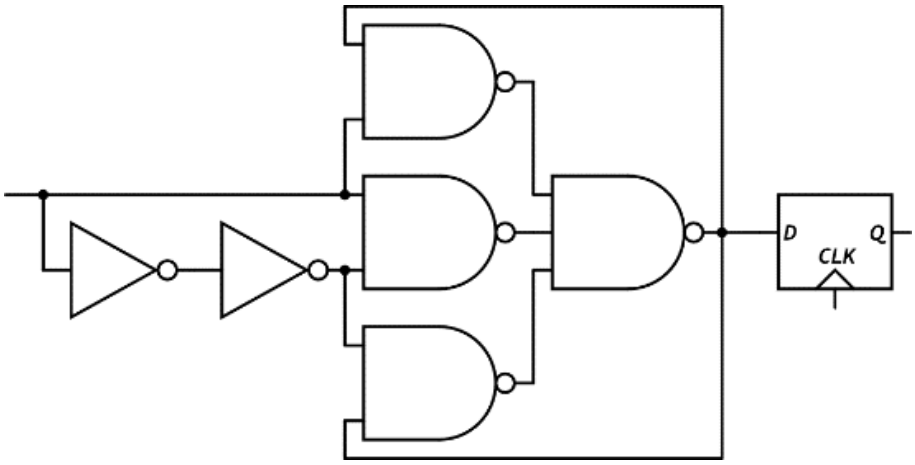


Figure 7.7: Example of inserted GG logic with filtering capability of 900 ps

7.2.5 SET-PAR: placement and routing tools for SET mitigation

A SET-aware Place & Route algorithm (SET-PAR) was developed for implementing SET hardened circuits on Flash-based FPGAs following the SGG mapper. In our case, the tool fits in our SEE analysis and mitigation flow as illustrated in Fig. 7.8.

The SET-PAR tools executes the placement and the routing algorithms in two distinct phases. The details of the two algorithms are depicted in the following subsections.

The PDD placement algorithm

The goal of the PDD placement algorithm is to find an effective placement for each logic node while optimizing the filtering capabilities of each logical path and minimizing the overhead delay of the circuit. The algorithm reads from the PDD circuit description the logic nodes, their interconnections and the I/O pins; while it loads the FPGA architectural characteristics, in terms of switch matrices rows and columns from the GAF FPGA array model. The core of the placement algorithm is based on an average Manhattan distance optimization technique including a detailed logical path analysis. The placement strategies consist in two phases, as it is illustrated in Fig. 7.9. After the initialization of the placement environment, the PDD placement firstly works on the optimal local placement of each logic cone optimizing the filtering capabilities in order to provide the maximal SET filtering; secondly it optimizes the overall logic cones placement locations for guaranteeing a routable solution.

In details, during the initialization phase, the circuit description is loaded into a directed graph called *L_element*, where each node is characterized by a type and an initial unplaced location and the set of input output nets (*io_nets*) is identified. Based on the PDD and GAF information, the placement area (P_Area) is generated. At the beginning of the first phase, it is computed the optimal filtering capabilities of each logic cone without considering the routing characteristics. This coefficient, named ΔK, provides the best SET filtering coefficient that a specific logic cones can achieve. The placement is then performed

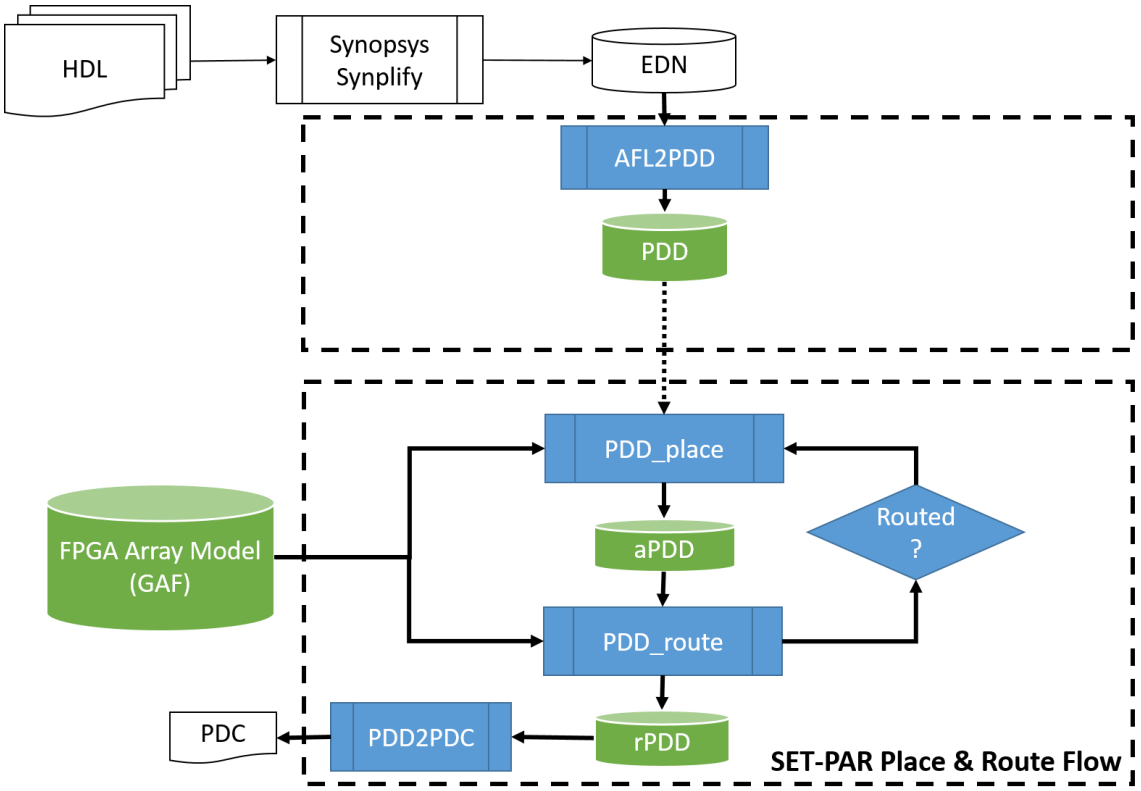


Figure 7.8: SET-aware Place & Route (SET-PAR) flow

analyzing the type of logic elements involved and each gate is characterized by a specific inverting coefficient which is calculated on the basis of the probability that the logic function provides in output the logical opposite value of input. The *Place_Manhattan_Min* function generates a temporal placement following: if a couple of connected gates have an inverting coefficient greater than 0.5, the placement is performed locating the two gates with a minimal distance; in the other cases, the placement is performed in a longer distance limited by the *local_k* coefficient. This process is repeated until the temporary placement has a SET coefficient (*SET_k*) which is equivalent to the optimal one with a degree of freedom provided by a flexibility coefficient *D*.

The second phase consists in the global placement; all the temporary placement macros are placed on P_area. During this phase, the major issue is due to the logic gates sharing multiple logic cones. These gates are identified and the placement modified accordingly minimizing the distance between their original positions with respect to the temporary placement. Finally, a new placement annotated PDD file (aPDD) including the placement location of each logic element within the FPGA array is generated.

```
PDD_place (PDD description)
{
      /*Preliminary Phase*/
      L_Elements (type, locations) = reading_logic(PDD)
      For each Logic ∈ L_Element
            io_net(Logic) = read_inout_logic (Logic);
      P_Area = create_placement_area (PDD, GAF);
      /*Local Logic Cone Placement, Phase 1*/
      Cone_Macro(i) = extract_cone(P_Area);
      K = optimal_SET(Macro_set);
      For each Cone_Macro(i){
            local_k = 0;
            do until (SET_k <= (K+D))
            {
                  For each Logic ∈ Cone_Macro(i)
                        Place_Macro = Place_Manhattan_Min (Logic, local_K);
                  (SET_k,D) = compute_SET_filters(Place_Macro);
                  If (SET_k < K) then local_K++;
            }
      }
      /*Global Placement, Phase 2*/
      do until (to_place_element == 0)
      {
            For each Place_Macro(i)
                  Shared_logic = Global_place(Place_Macro(i));
                  Place_Area(Shared_Logic, Place_Macro(i));
      }
      create_PDD(P_area);
}
```

Figure 7.9: The PDD placement algorithm

7.3 Experiment results and analysis

The proposed design flow has been experimentally evaluated in order to prove its efficiency and accuracy. Two experimental analyses have been performed. The former aimed at evaluating the improvements of the SET mitigation on transient error pulses set ranging from 10ps to 10ns. The latter consists of a detailed timing analysis measuring the maximum delay of the critical path between the different solutions of the implemented circuits. Both the evaluations have been performed using some benchmark circuits included in the ITC99 benchmarks and including a RISC processor core [18]. Besides, all the circuits have been implemented using a Microsemi A3P250 Flash-based FPGA based on 130 nm CMOS manufacturing process and counting up to 6,144 logic cells [43].

The characteristics of the implemented circuits are illustrated in Table 7.1, where for each circuit is reported the number of logic cells, the number of nets and the number of routing segments including short and long wires, obtained with commercial tools and with the proposed SET-PAR approach.

Table 7.1: Characteristics of the implemented benchmark circuits

Circuit	Logic Cells [#]	Routing Nets [#]	Commercial tools and [69]		SET-PAR approach	
			Short Wires [#]	Long Lines [#]	Short Wires [#]	Long Lines [#]
B03	131	466	1,365	555	1,112	432
B05	544	2,070	6,407	2,573	6,301	2,509
B07	244	1,038	2,821	1,066	2,373	893
B08	149	464	1,326	545	1,199	486
B09	115	480	1,705	663	1,348	535
B10	168	474	1,375	543	1,251	486
B11	311	1,290	3,798	1,423	3,094	1,197
B12	619	2,856	7,731	2,905	6,916	2,404
B13	222	584	1,429	516	1,203	450
B14	3,872	16,722	71,621	27,025	70,517	26,636
RISC	3,425	11,814	36,368	13,112	33,642	11,951

As it is possible to observe the physical routing used by the SET-PAR algorithm provides a number of routing segments ranging from 10% to 20% lower than the one obtained using commercial solutions based on the previously developed method [69].

The analysis of the Single Event Transient sensitivity has been performed with the Single Event Transient Analyzer platform [70] using a large set of transient pulses ranging from 10 ps to 10 ns thus covering the entire realistic transient events induced in a harsh radiation environment. The validation has been obtained by randomly choose the location where to electrically inject the pulses and by applying random input stimuli to the tested circuits. Then, a result was classified as wrong answer, if during the test application produces at least one output pattern that differs from the expected one. Finally, the number of SETs provoking errors on the circuit outputs were counted. The results gathered are illustrated in Fig. 7.10, where the reduction was indicated in percentage, of the SET number provoking errors within the output of the implemented circuits. The reduction is extremely high since around 30% of SETs are correctly mitigated with respect to the previously developed approach. Besides, it is possible to notice that the SET-PAR tool works properly independently from the complexity of the implemented circuit (e.g., the overall SET-induced errors reduction is always greater than 27% whatever is the circuit under analysis).

Finally, the benchmark circuits have been timing analyzed using Microsemi vendor's software in order to estimate the maximum working frequency. The results obtained with the timing analysis are reported on Fig. 7.10 as percentage of frequency improvement provided by the SET-PAR tools with respect to the previous solution. The results demonstrate that the SET-PAR algorithm is efficiently implementing circuits on Flash-based

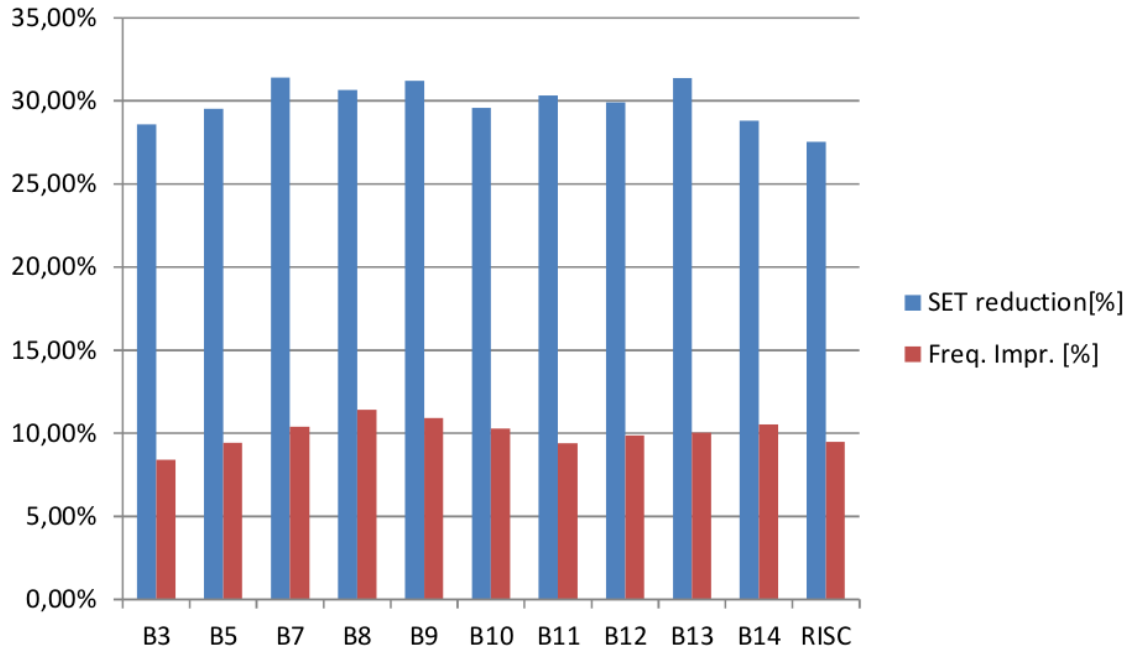


Figure 7.10: SET reduction and Frequency improvements of the SET-PAR implemented circuits w.r.t. the previously developed solution based on Microsemi commercial tools

FPGA.

7.3.1 Radiation experiment on Microsemi Flash-based FPGA

Besides the analysis performed above, a radiation experiment was also carried out with a heavy ions beam in order to provoke radiation-induced SET and classify their effects.

Experiment setup

RISC5X from OpenCores is a RISC CPU that is compatible with the 12-bit opcode PIC family [73]. It consists of an Instruction Decoder (IDEC), an Arithmetic Logic Unit (ALU) which is capable of 8-bit addition, subtraction and logic shift operation, and a register file of 128 bytes used as RAM. The RISC5x has three 8-bit wide ports which can be used as input and output ports connecting to different peripherals. The ports are named PORTA, PORTB and PORTC respectively, as illustrated in Fig. 7.11. To map RISC5x to the FPGA, a ROM module was added to hold the instructions by means of signal array in VHDL, which was inferred as logic gates instead of Flip-Flops or latches by the Synplify tool in Libero IDE from Microsemi.

The original RISC5x from OpenCores has no fault tolerant strategy applied. In order to reduce data corruptions in RAM caused by SEUs and to focus on the analysis of SETs induced by the radiation particles, the original RAM module was replaced with another version protected by a Hamming Code which is enabled to correct two bit errors when

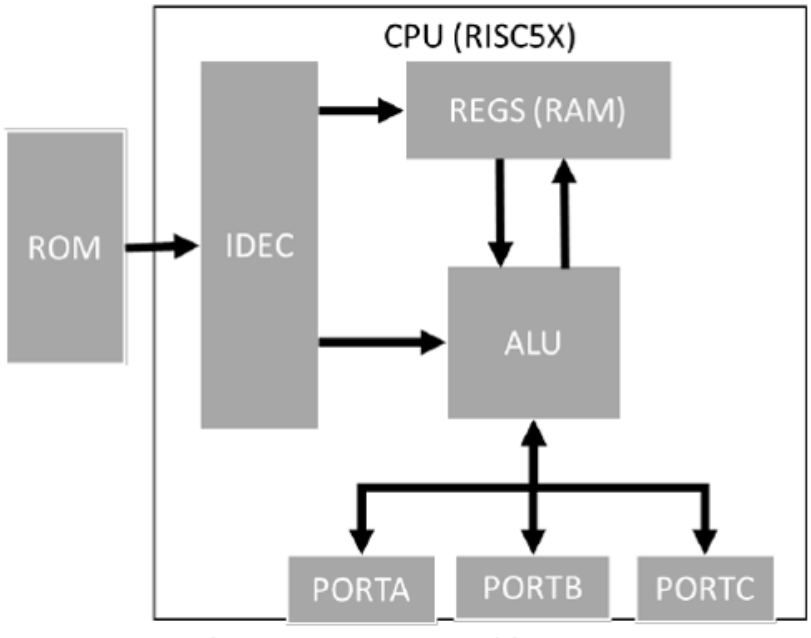


Figure 7.11: RISC5x architecture

they reside separately in the higher and lower half of the 8-bit register. The scheme of the Error Correction Code (ECC) applied to the register file RAM resources is depicted in Fig. 7.12.

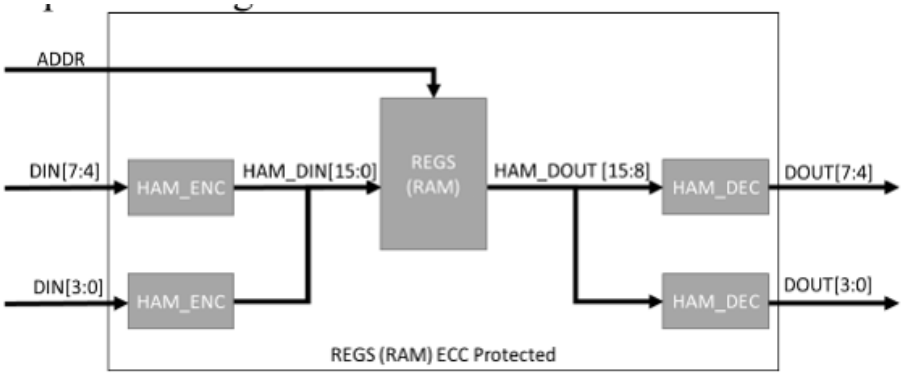


Figure 7.12: ECC scheme adopted in order to protect RISC register file, implemented using Flash-based FPGA embedded RAM modules against SEU accumulation

A hardened implementation of the RISC microprocessor described above was developed, applying TMR at different logic levels in order to drastically mitigate SEUs. Two different methods were used. The first method consists in applying the Synplify tool oriented to radiation hardened FPGA designs [44]; it applies the triplication of all the Flip-Flops inserting a majority voter on their outputs. The second has been applied at

the entity level, in which the components in RISC5x, such as IDEC and ALU are directly triplicated as represented in Fig. 7.13, except for the register file module (REGS) which has been already protected by ECC implementation as mentioned above.

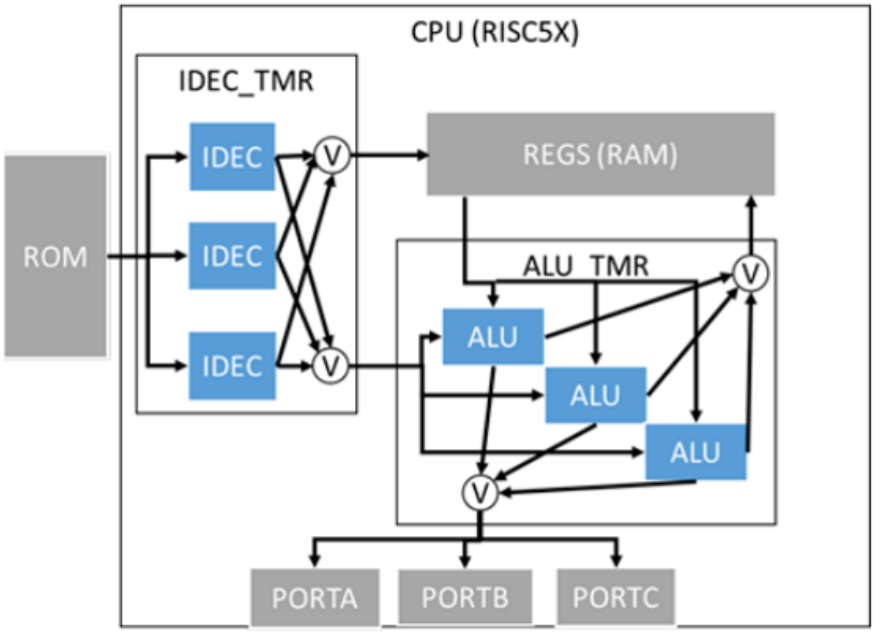


Figure 7.13: TMR at entity level (IDEC & ALU)

Totally four versions of the design were prepared and tested in the radiation experiment for comparison, whose characteristics are reported in Table 7.2. The four versions are:

1. *RISC5x* is the original version with only ECC protection applied to the register file memory.This version is also called Plain version.
2. *RISC5x TMR+GG(1ns)* is based on the Plain version with entity level TMR applied and also with Guard Gates targeting on SET filtering up to pulse with of 1 ns.
3. *RISC5x TMR-FF* is based on the Plain version with Synplify's TMR technique applied on the FFs in the design.
4. *RISC5x SEE-aware P&R* is the version with proposed SEE analysis and mitigation flow applied on the Plain version.

The same test program was used running on the RISC5x micro-controller across the four different versions, which generates a sequence of data output with fixed time gap between each output. During execution of the test program, SEEs can cause errors in different components in FPGA, which can lead to corrupted data outputs and wrong time gaps between outputs. At this step, still there are errors that will be silenced by the fault tolerant strategies applied or the initialization stage in the test program.

Table 7.2: Characteristics of four RISC5x versions

Design version	Logic Gates [#]	FFs [#]
RISC5x	1,401	1,156
RISC5x TMR+GG(1ns)	20,808	3,468
RISC5x TMR-FF	2,403	3,468
RISC5x SEE-aware P&R	5,514	3,468

Another FPGA board from Altera was used as monitor board to control the Microsemi board, capture the data outputs and communicate with the host PC outside the radiation beam chamber, as illustrated in the scheme reported in Fig. 7.14. As the three ports in RISC5x were synchronized with clock source in the Microsemi board which was not shared with the Altera board, another port (PORTB) was used to indicate valid data on PORTA so that the asynchronous capturer in the monitor board can read the correct data from PORTA and store them in SRAM. The monitor board also controls the reset signal of the Microsemi board, and to avoid reset signal hazard, the reset signal was triplicated and used a voter in the design to ensure that the test program will be executed and stopped, as wanted.

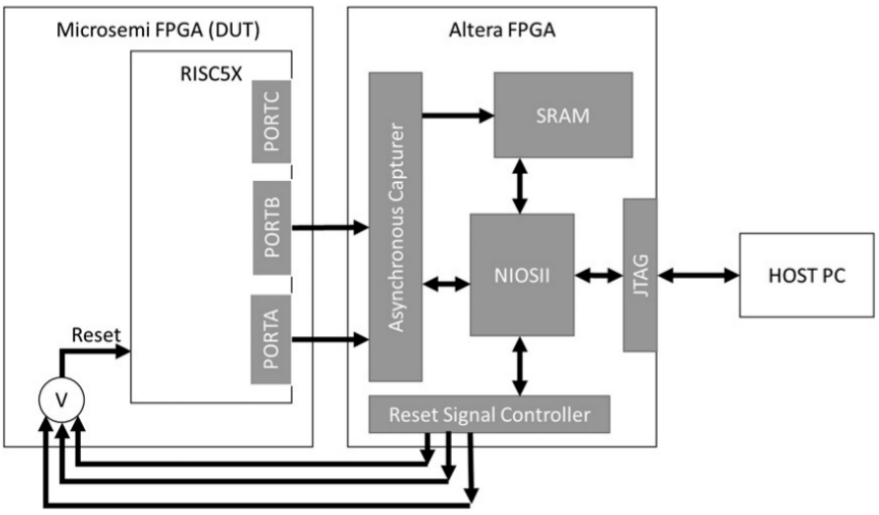


Figure 7.14: Radiation experiment setup

The radiation campaign has been performed in September 2013 at the Cyclotron of the Université Catholique de Louvain (UCL) on four different versions of the RISC5x micro-processor. The RISC working frequency has been settled to 20 MHz while the experiments have been performed using a Kripton ion with a fluence of $3.04e8$ (particles) and an average flux of $1e4$ (particles/sec).

Radiation experiment results and analysis

The experimental results have been collected comparing the cross-section (errors/particles) for each of the implemented RISC5x version. The results obtained during the radiation test campaign demonstrated that the RISC5x implemented with SEE-aware analysis and mitigation flow has a sensitivity of about two orders of magnitude with respect to the RISC hardened with full TMR and Guard-Gate filtering 1 ns SET pulse. Error bars were computed on the basis of the Monte Carlo process, which is calculated as $\frac{1}{\sqrt{N}}$ where N corresponds to the number of errors counted. In the worst case, the statistical error is 10%; the obtained results are reported in Fig. 7.15.

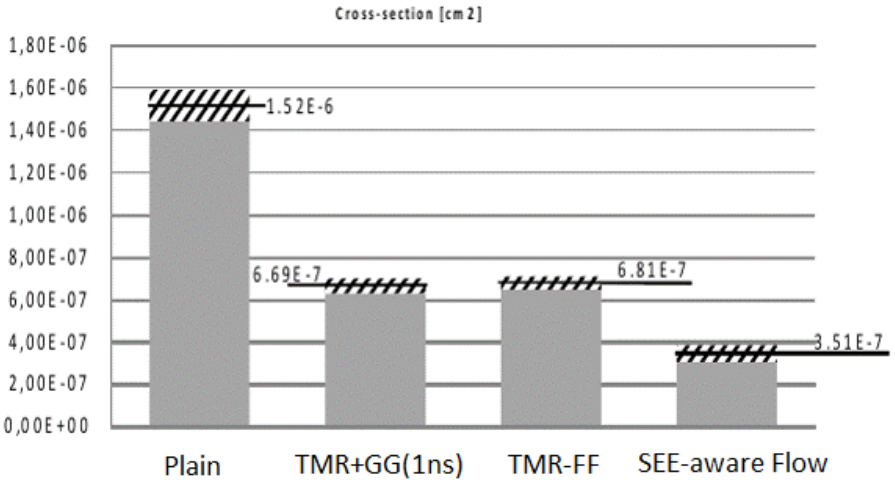


Figure 7.15: SEE cross-section comparison between different RISC5x versions

The Plain version of the RISC only protected with the ECC mechanism applied to the register file memory, reports a cross-section of $1.52e-6cm^2$. The TMR implementations, either using Guard-Gates (TMR+GG) or adopting TMR on sequential element (TMR-FF) have a reduction of cross-section of about 55% versus the plain version, while a negligible difference between the two TMR solutions is observed. On the other hand, the RISC5x implemented and mapped using the proposed SEE-aware flow provides a reduction of more than 78% with respect to the plain version and about 48% less SEE sensibility compared to the TMR RISC5x implementations. Please note that performances are not degraded among the execution of three different RISC5x versions. In order to characterize the collected data, the data outputs provided by the RISC5x from the PORTA was analyzed, which can be divided into sequences containing 8 data output records each. The time difference between two adjacent data records in a sequence was measured by the monitor board and reports a fixed number of clock cycles when no error is presented in the design. According to the data outputs collected by the monitor board, the errors caused by SEEs can be functionally classified into 4 types:

1. *time gap*, corresponding to the time difference between two data output outside the

margin of the corrected ones;

2. *wrong data*, corrupted data reported on the PORTA output;
3. *wrong records*, an erroneous number of records in a data sequence;
4. *wrong sequence*, an erroneous number of data sequences.

The classification, reported in Fig. 7.16, gives insight on the improvements performed by the SEE aware placement design of the RISC5x. In particular, the SEE-aware RISC5x reports a drastic reduction of the wrong data effect (i.e., more than 60% versus the plain RISC5x) and an evident reduction of about 25% and 30% versus the TMR implementations, related to the time gap and missing sequence effects.

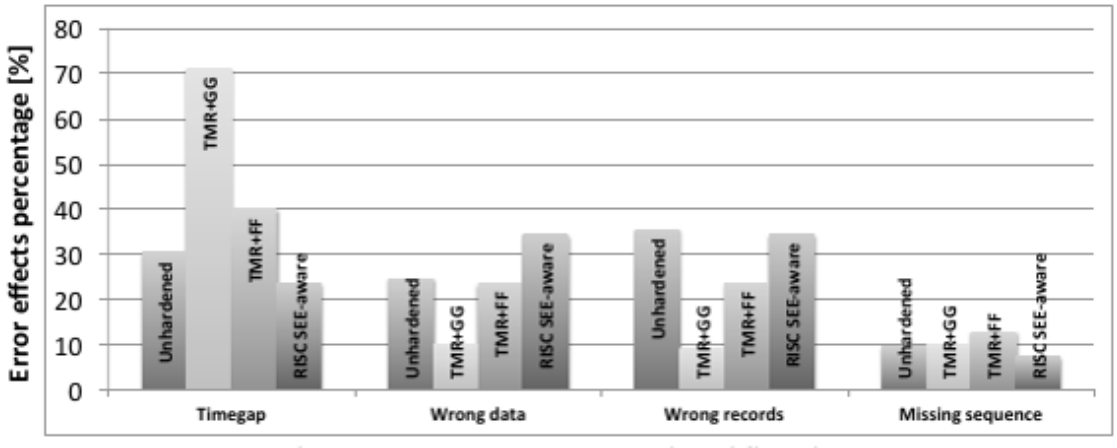


Figure 7.16: Error events classification

In this work, a novel design flow is presented for implementing circuits hardened versus SEE effects on Flash-based FPGAs. The design flow contains an accurate SET analytical model, a FPGA logic and route model, the SETA (SET Analyzer) tool and SET-PAR (SET-aware Place And Route) tool integrated together. Since The design flow takes the netlist and place constraint files from standard commercial tools (in our case, Microsemi Libero), modifies the netlist and generates the constraints file so that it can be integrated with commercial toolchain easily.

Verified by analysis and radiation experiment, the solution proposed here has the main advantage of being capable of fully mitigate SEUs and drastically reduce the impact of SETs with a limited number of overhead resources with respect to traditional redundancy and filtering-based approaches. In particular our approach presents a reduction of the area overhead of more than 83% with respect to traditional mitigation approaches, while circuits results two orders of magnitude less sensitive with respect to soft errors. As future research, a plan has been made to perform further evaluation on the performance of the proposed method addressing high performance computation circuits implemented on ultra-nanometer Flash-based technology.

Last but not least, the SET analytical model and the SET-aware Place & Route tool developed are not bound to the FPGA design, instead, with input of cell technology library and GDS-II layout model of the target design, our design flow can be tuned to work with ASIC design for SEE-aware analysis and mitigation solution. And this is also part of our current and future works.

References

[1] Francesco Abate, Luca Sterpone, Massimo Violante, and F Lima Kastensmidt. "A study of the single event effects impact on functional mapping within flash-based FPGAs". In: *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.* IEEE. 2009, pages 1226–1229.

[2] Igor Aleksejev, Artur Jutman, Sergei Devadze, Sergei Odintsov, and Thomas Wenzel. "FPGA-based synthetic instrumentation for board test". In: *Test Conference (ITC), 2012 IEEE International*. IEEE. 2012, pages 1–10.

[3] Zeyad Alkhalifa, VS Sukumaran Nair, Narayanan Krishnamurthy, and Jacob A. Abraham. "Design and evaluation of system-level checks for on-line control flow error detection". In: *Parallel and Distributed Systems, IEEE Transactions on* 10.6 (1999), pages 627–641.

[4] ARM. URL: http://www.arm.com/products/system-ip/debug-trace/.

[5] ARM. *CoreSight Program Flow Trace (PFTv1.0 and PFTv1.1) Architecture Specification*. URL: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0035b/IHI0%20035B_cs_pft_v1_1_architecture_spec.pdf.

[6] Jose Rodrigo Azambuja, Mauricio Altieri, Jurgen Becker, and Fernanda Lima Kastensmidt. "HETA: hybrid error-detection technique using assertions". In: *Nuclear Science, IEEE Transactions on* 60.4 (2013), pages 2805–2812.

[7] José Rodrigo Azambuja, Ângelo Lapolli, Lucas Rosa, and Fernanda Lima Kastensmidt. "Detecting SEEs in microprocessors through a non-intrusive hybrid technique". In: *Nuclear Science, IEEE Transactions on* 58.3 (2011), pages 993–1000.

[8] José Rodrigo Azambuja, Samuel Pagliarini, Mauricio Altieri, Fernanda Lima Kastensmidt, Michael Hübner, Jürgen Becker, Gilles Foucard, and Raoul Velazco. "A fault tolerant approach to detect transient faults in microprocessors based on a non-intrusive reconfigurable hardware". In: *Nuclear Science, IEEE Transactions on* 59.4 (2012), pages 1117–1124.

[9] José Rodrigo Azambuja, Samuel Pagliarini, Lucas Rosa, and Fernanda Lima Kastensmidt. "Exploring the limitations of software-based techniques in SEE fault coverage". In: *Journal of Electronic Testing* 27.4 (2011), pages 541–550.

[10] T Baghai and V Hildeman. "Avionics Certification: A Complete Guide to DO-178 (Software), DO-254 (Hardware)". In: *United States* (2007).

[11] Niccolo' Battezzati, S Gerardin, A Manuzzato, D Merodio, A Paccagnella, C Poivey, Luca Sterpone, and Massimo Violante. "Methodologies to study frequency-dependent single event effects sensitivity in flash-based FPGAs". In: *Nuclear Science, IEEE Transactions on* 56.6 (2009), pages 3534–3541.

[12] Mark P Baze, Steven P Buchner, and Dale McMorrow. "A digital CMOS design technique for SEU hardening". In: *Nuclear Science, IEEE Transactions on* 47.6 (2000), pages 2603–2608.

[13] Alfredo Benso, Stefano Di Carlo, Giorgio Di Natale, and Paolo Prinetto. "A watchdog processor to detect data and control flow errors". In: *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*. IEEE. 2003, pages 144–148.

[14] Melanie Berg, H Kim, M Friendlich, C Perez, C Seidleck, K LaBel, and R Ladbury. "SEU analysis of complex circuits implemented in Actel RTAX-S FPGA devices". In: *IEEE Trans. Nucl. Sci* 58.3 (2011), pages 1015–1022.

[15] S. Bergaoui, P. Vanhauwaert, and R. Leveugle. "IDSM: An improved disjoint signature monitoring scheme for processor behavioral checking". In: *Test Workshop - LATW, 2014 15th Latin American*. Mar. 2014, pages 1–6. DOI: 10.1109/LATW.2014.6841915.

[16] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for deep-submicron FPGAs*. Volume 497. Springer Science & Business Media, 2012.

[17] Phillipe Cheynet, Bogdan Nicolescu, Raoul Velazco, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. "Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors". In: *IEEE Transactions on Nuclear Science* 47.6 (2000), pages 2231–2236.

[18] F. Corno, M. S. Reorda, and G. Squillero. "RT-level ITC'99 benchmarks and first ATPG results". In: *IEEE Design Test of Computers* 17.3 (July 2000), pages 44–53. ISSN: 0740-7475. DOI: 10.1109/54.867894.

[19] Paul E Dodd and Lloyd W Massengill. "Basic mechanisms and modeling of single-event upset in digital microelectronics". In: *Nuclear Science, IEEE Transactions on* 50.3 (2003), pages 583–602.

[20] B. Du, M. S. Reorda, L. Sterpone, L. Parra, M. Portela-Garcia, A. Lindoso, and L. Entrena. "Exploiting the debug interface to support on-line test of control flow errors". In: *On-Line Testing Symposium (IOLTS), 2013 IEEE 19th International*. July 2013, pages 98–103. DOI: 10.1109/IOLTS.2013.6604058.

[21] B. Du, M. Sonza Reorda, L. Sterpone, L. Parra, M. Portela-Garcia, A. Lindoso, and L. Entrena. "A new solution to on-line detection of Control Flow Errors". In: *On-Line Testing Symposium (IOLTS), 2014 IEEE 20th International*. July 2014, pages 105–110. DOI: 10.1109/IOLTS.2014.6873680.

[22] B. Du, M. Sonza Reorda, L. Sterpone, L. Parra, M. Portela-Garcia, A. Lindoso, and L. Entrena. "On-line Test of Control Flow Errors: A new Debug Interface-based approach". In: *IEEE Transactions on Computers* PP.99 (2015), pages 1–1. ISSN: 0018-9340. DOI: 10.1109/TC.2015.2456014.

[23] Heidrun Engel. “Data flow transformations to detect results which are corrupted by hardware faults”. In: *High-Assurance Systems Engineering Workshop, 1996. Proceedings., IEEE*. IEEE. 1996, pages 279–285.

[24] Luis Entrena, Mario Garcia-Valderas, Raul Fernandez-Cardenal, Almudena Lindoso, Marta Portela, and Celia Lopez-Ongil. “Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection”. In: *Computers, IEEE Transactions on* 61.3 (2012), pages 313–322.

[25] Luis Entrena, Mario García Valderas, Raúl Fernández Cardenal, Marta Portela García, and Celia López Ongil. “SET emulation considering electrical masking effects”. In: *IEEE Transactions on Nuclear Science* 4.56 (2009), pages 2021–2025.

[26] Hongxia Fang, Zhiyuan Wang, Xinli Gu, and Krishnendu Chakrabarty. “Mimicking of functional state space with structural tests for the diagnosis of board-level functional failures”. In: *Test Symposium (ATS), 2010 19th IEEE Asian*. IEEE. 2010, pages 421–428.

[27] Aeroflex Gaisler. “Leon3 processor”. In: *Nanoscale Integration and Modeling (NIMO) Group* (2010).

[28] Jiri Gaisler, Sandi Habinc, and E Catovic. “Grlib ip library user’s manual”. In: *Aeroflex Gaisler* (2010).

[29] Michelangelo Grosso, M Sonza Reorda, Marta Portela-García, Mario García-Valderas, Celia López-Ongil, and Luis Entrena. “An on-line fault detection technique based on embedded debug features”. In: *On-Line Testing Symposium (IOLTS), 2010 IEEE 16th International*. IEEE. 2010, pages 167–172.

[30] LMOSS Hangout and S Jan. “The minimips project”. In: (2009). URL: http://opencores.org/project,minimips.

[31] Jonathan Heiner, Benjamin Sellers, Michael Wirthlin, and Jeff Kalb. “FPGA partial reconfiguration via configuration scrubbing”. In: *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. IEEE. 2009, pages 99–104.

[32] Jian Huang and David J Lilja. “Exploiting basic block value locality with block reuse”. In: *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*. IEEE. 1999, pages 106–114.

[33] IEC61508 IEC. “61508 functional safety of electrical/electronic/programmable electronic safety-related systems”. In: *International electrotechnical commission* (1998).

[34] *ISO/DIS 26262-1 - Road vehicles â Functional safety â Part 1 Glossary*. Technical report. Geneva, Switzerland, July 2009.

[35] Mostafa Jafari-Nodoushan, Seyed Ghassem Miremadi, and Alireza Ejlali. “Control-flow checking using branch instructions”. In: *Embedded and Ubiquitous Computing, 2008. EUC’08. IEEE/IFIP International Conference on*. Volume 1. IEEE. 2008, pages 66–72.

[36] Artjom Jasnetski, Jaan Raik, Anton Tsertov, and Raimund Ubar. "New Fault Models and Self-Test Generation for Microprocessors Using High-Level Decision Diagrams". In: *Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2015 IEEE 18th International Symposium on*. IEEE. 2015, pages 251–254.

[37] Artur Jutman. "Filling a Gap in Board-Level At-Speed Test Coverage". In: *Defects, Adaptive Test, Yield and Data Analysis (DATA'2015), IEEE International Workshop On*. IEEE. 2015, pages 1–7.

[38] Artur Jutman, Sergei Devadze, Igor Aleksejev, and Thomas Wenzel. "Embedded synthetic instruments for Board-Level testing". In: *2012 17TH IEEE EUROPEAN TEST SYMPOSIUM (ETS)*. 2012.

[39] Artur Jutman, M Sonza Reorda, and H-J Wunderlich. "High Quality System Level Test and Diagnosis". In: *Test Symposium (ATS), 2014 IEEE 23rd Asian*. IEEE. 2014, pages 298–305.

[40] Markus Kowarschik and Christian Weiß. "An overview of cache optimization techniques and cache-aware numerical algorithms". In: *Algorithms for Memory Hierarchies*. Springer, 2003, pages 213–232.

[41] Erik Larsson, Bill Eklow, Scott Davidsson, Rob Aitken, Artur Jutman, and Christophe Lotz. "No Fault Found: The Root Cause". In: *VLSI Test Symposium (VTS), 2015 IEEE 33rd*. IEEE. 2015, pages 1–1.

[42] Wassim Mansour and Raoul Velazco. "An automated SEU fault-injection method and tool for HDL-based designs". In: *Nuclear Science, IEEE Transactions on* 60.4 (2013), pages 2728–2733.

[43] Microsemi. "Automotive ProASIC3 Flash Family FPGAs". In: *Datasheet, Revision 5* (2013).

[44] Microsemi. "Using Synplify to Design in Microsemi Radiation-Hardened FPGAs". In: *Application Note AC139* (2012).

[45] Subhasish Mitra, Norbert Seifert, Ming Zhang, Quan Shi, and Kee Sup Kim. "Robust system design with built-in soft-error resilience". In: *Computer* 2 (2005), pages 43–52.

[46] Shubhendu S Mukherjee, Christopher Weaver, Joel Emer, Steven K Reinhardt, and Todd Austin. "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor". In: *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society. 2003, page 29.

[47] "NanGate FreePDK45 Open Cell Library". In: *Available at http://www.nangate.com/?page_id=2325* (2011).

[48] Gabriel L Nazar, Leonardo P Santos, and Luigi Carro. "Accelerated FPGA repair through shifted scrubbing". In: *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE. 2013, pages 1–6.

[49] B Nicolescu, Y Savaria, and R Velazco. “Software detection mechanisms providing full coverage against single bit-flip faults”. In: *Nuclear Science, IEEE Transactions on* 51.6 (2004), pages 3510–3518.

[50] B Nicolescu and Raoul Velazco. “Detecting soft errors by a purely software approach: method, tools and experimental results”. In: *Embedded Software for SoC*. Springer, 2003, pages 39–51.

[51] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. “Control-flow checking by software signatures”. In: *Reliability, IEEE Transactions on* 51.1 (2002), pages 111–122.

[52] L. Parra, A. Lindoso, M. Portela-Garcia, L. Entrena, B. Du, M. S. Reorda, and L. Sterpone. “A New Hybrid Nonintrusive Error-Detection Technique Using Dual Control-Flow Monitoring”. In: *IEEE Transactions on Nuclear Science* 61.6 (Dec. 2014), pages 3236–3243. ISSN: 0018-9499. DOI: 10.1109/TNS.2014.2361953.

[53] Lucas Parra, Almudena Lindoso, Marta Portela, Luis Entrena, Felipe Restrepo-Calle, Sergio Cuenca-Asensi, and Antonio Martínez-Álvarez. “Efficient mitigation of data and control flow errors in microprocessors”. In: *Nuclear Science, IEEE Transactions on* 61.4 (2014), pages 1590–1596.

[54] Luis Parra, Almudena Lindoso, Michelangelo Grosso, M Sonza Reorda, Marta Portela-García, Mario García-Valderas, Celia López-Ongil, and Luis Entrena. “Control flow checking through embedded debug interface”. In: *Design of Circuits and Integrated Systems (DCIS), 26th Conference on*. 2011, pages 339–342.

[55] J Perez Acle, R Cantoro, AT Hailemichael, E Sanchez, and M Sonza Reorda. “Observability solutions for in-field functional test of processor-based systems”. In: *Design of Circuits and Integrated Systems (DCIS), 2015 Conference on*. IEEE. 2015, pages 1–6.

[56] Marta Portela-García, Michelangelo Grosso, M Gallardo-Campos, M Sonza Reorda, Luis Entrena, Mario García-Valderas, and Celia López-Ongil. “On the use of embedded debug features for permanent and transient fault resilience in microprocessors”. In: *Microprocessors and Microsystems* 36.5 (2012), pages 334–343.

[57] Mihalis Psarakis, Dimitris Gizopoulos, Ernesto Sanchez, and Matteo Sonza Reorda. “Microprocessor software-based self-testing”. In: *IEEE Design & Test of Computers* 3 (2010), pages 4–19.

[58] Heather M Quinn, Dolores A Black, William H Robinson, and Stephen P Buchner. “Fault simulation and emulation tools to augment radiation-hardness assurance testing”. In: *Nuclear Science, IEEE Transactions on* 60.3 (2013), pages 2119–2142.

[59] Maurizio Rebaudengo, Matteo Sonza Reorda, Marco Torchiano, and Massimo Violante. “Soft-error detection through software fault-tolerance techniques”. In: *Defect and Fault Tolerance in VLSI Systems, 1999. DFT’99. International Symposium on*. IEEE. 1999, pages 210–218.

[60] Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. "Software-level soft-error mitigation techniques". In: *Soft Errors in Modern Electronic Systems*. Springer, 2011, pages 253–285.

[61] Paolo Rech, Caroline Aguiar, R Ferreira, Christopher Frost, and Luigi Carro. "Neutron radiation test of graphic processing units". In: *On-Line Testing Symposium (IOLTS), 2012 IEEE 18th International*. IEEE. 2012, pages 55–60.

[62] Sana Rezgui, JJ Wang, Yinming Sun, Brian Cronquist, and John McCollum. "Configuration and routing effects on the SET propagation in flash-based FPGAs". In: *Nuclear Science, IEEE Transactions on* 55.6 (2008), pages 3328–3335.

[63] Sana Rezgui, Raymond Won, and Jonathan Tien. "Set characterization and mitigation in 65-nm cmos test structures". In: *Nuclear Science, IEEE Transactions on* 59.4 (2012), pages 851–859.

[64] Andreas Riefert, Riccardo Cantoro, Matthias Sauer, Matteo Sonza Reorda, and Bernd Becker. "On the automatic generation of SBST test programs for in-field test". In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium. 2015, pages 1186–1191.

[65] Andreas Riefert, Lyl Ciganda, Matthias Sauer, Paolo Bernardi, M Sonza Reorda, and Bernd Becker. "An effective approach to automatic functional processor test generation for small-delay faults". In: *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. IEEE. 2014, pages 1–6.

[66] Ricardo Santos, Shyamsundar Venkataraman, Aruneema Das, and Ajit Kumar. "Criticality-aware scrubbing mechanism for SRAM-based FPGAs". In: *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*. IEEE. 2014, pages 1–8.

[67] Alodeep Sanyal, Krishnendu Chakrabarty, Mahmut Yilmaz, and Hideo Fujiwara. "RT-level design-for-testability and expansion of functional test sequences for enhanced defect coverage". In: *Test Conference (ITC), 2010 IEEE International*. IEEE. 2010, pages 1–10.

[68] L. Sterpone and N. Battezzati. "On the mitigation of SET broadening effects in integrated circuits". In: *Design and Diagnostics of Electronic Circuits and Systems (DDECS), 2010 IEEE 13th International Symposium on*. Apr. 2010, pages 36–39. DOI: 10.1109/DDECS.2010.5491820.

[69] L. Sterpone and Boyang Du. "Analysis and mitigation of single event effects on flash-based FPGAS". In: *Test Symposium (ETS), 2014 19th IEEE European*. May 2014, pages 1–6. DOI: 10.1109/ETS.2014.6847804.

[70] Luca Sterpone, Niccolo' Battezzati, and V Ferlet-Cavrois. "Analysis of SET propagation in flash-based FPGAs by means of electrical pulse injection". In: *Nuclear Science, IEEE Transactions on* 57.4 (2010), pages 1820–1826.

[71] Luca Sterpone, Niccolo' Battezzati, F Lima Kastensmidt, and Raul Chipana. "An analytical model of the propagation induced pulse broadening (PIPB) effects on single event transient in flash-based FPGAs". In: *Nuclear Science, IEEE Transactions on* 58.5 (2011), pages 2333–2340.

[72] "The Nexus 5001 forum Standard for a Global Embedded Processor Debug Interface". Version 2.0. In: *IEEE-ISTO 5001-2003* (2003).

[73] "The RISC5x project". In: (2009). URL: http://opencores.org/project,risc5x.

[74] Xilinx UG190. "Virtex-5 FPGA user guide". In: *UG190* 5 (2009).

[75] Ramtilak Vemu and Jacob A Abraham. "Ceda: Control-flow error detection through assertions". In: *On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International*. IEEE. 2006, 6–pp.

[76] Ramtilak Vemu, Sankar Gurumurthy, and Jacob A Abraham. "ACCE: Automatic correction of control-flow errors". In: *Test Conference, 2007. ITC 2007. IEEE International*. IEEE. 2007, pages 1–10.

[77] JJ Wang, S Samiee, H-S Chen, C-K Huang, M Cheung, J Borillo, S-N Sun, B Cronquist, and J McCollum. "Total ionizing dose effects on flash-based field programmable gate array". In: *Nuclear Science, IEEE Transactions on* 51.6 (2004), pages 3759–3766.

[78] Zhen Wang, Mark Karpovsky, and Ajay Joshi. "Reliable MLC NAND flash memories based on nonlinear t-error-correcting codes". In: *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*. IEEE. 2010, pages 41–50.

[79] Gilson I Wirth, Michele G Vieira, Egas Henes Neto, and FGL Kastensmidt. "Single event transients in combinatorial circuits". In: *Integrated Circuits and Systems Design, 18th Symposium on*. IEEE. 2005, pages 121–126.

[80] Carmichael C XAPP197. *Triple Modular Redundancy Design Techniques for Virtex FPGAs*. 2006.

[81] Xilinx Application Notes XAPP216. *Correcting Single-Event Upset Through Virtex Partial Reconfiguration*. 2000.

[82] Wenyao Xu, Jia Wang, Yu Hu, Ju-Yueh Lee, Fang Gong, Lei He, and Majid Sarrafzadeh. "In-place FPGA retiming for mitigation of variational single-event transient faults". In: *Circuits and Systems I: Regular Papers, IEEE Transactions on* 58.6 (2011), pages 1372–1381.